UNIVERSITÉ DE MONTRÉAL

IMPLÉMENTATION D'UN MODÈLE DE COMMUNICATION TRANSACTIONNEL DANS UNE PLATE-FORME EN SYSTEMC

OLIVIER BENNY
DÉPARTEMENT DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE ÉLECTRIQUE)
JANVIER 2004

Your file Votre référence
ISBN: 0-612-89179-8
Our file Notre référence
ISBN: 0-612-89179-8

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

IMPLÉMENTATION D'UN MODÈLE DE COMMUNICATION TRANSACTIONNEL DANS UNE PLATE-FORME EN SYSTEMC

---

présenté par: BENNY Olivier

en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen constitué de:

Mme NICOLESCU Gabriela, Doctorat, présidente

M. BOIS Guy, Ph.D., membre et directeur de recherche

M. BOYER François-Raymond, Ph.D., membre et codirecteur de recherche

M. KHOUAS Abdelhakim, Doctorat, membre

à Geneviève

REMERCIEMENTS

Je tiens d'abord à remercier mon directeur de recherche, Guy Bois, pour sa perspicacité à propos du projet et sa confiance qu'il a portée envers moi.

Une mention spéciale va à mon codirecteur, François-Raymond Boyer, ainsi qu'au professeur El Mostapha Aboulhamid, pour avoir partagé leur expérience et beaucoup de leur temps au soutien de mes travaux.

Au sein du groupe, je remercie plusieurs collègues qui m'ont aidé de près ou de loin, soit Bruno Lavigueur, Mame Maria Mbaye, Luc Filion, Marc Bertola, François Deslauriers et Anh Tuan Nguyen.

Finalement, ce projet n'aurait pas de sens sans Mathieu Rondonneau et Jérôme Chevalier. Je vous dois une bonne partie de ce mémoire ainsi que de bons moments passés ensemble.

RÉSUMÉ

Au niveau système, la conception d'une application embarquée peut être amorcée en premier lieu par l'élaboration d'un modèle purement fonctionnel, où l'on exprime la fonctionnalité désirée d'une application en termes de modules, de ports, d'interfaces, de processus et de canaux.

Un projet actuel au sein du groupe de recherche en microélectronique de l'École Polytechnique de Montréal a pour but la conception d'une plate-forme en SystemC (une bibliothèque d'objets en C++ qui permet de modéliser à haut niveau et de simuler des systèmes matériels et logiciels) qui pourra servir à la fois d'architecture de base et d'outil d'aide à la conception des systèmes numériques. L'objectif primordial de notre méthodologie est de permettre aux concepteurs de profiter pleinement de la plate-forme pour pouvoir développer leurs applications, sans avoir à modifier le système d'exploitation ou les composants généraux de la plate-forme. Cette dernière facilite l'étape du partitionnement des modules ; c'est-à-dire de décider de la nature matérielle ou logicielle que prendra ces différents modules lors de la synthèse.

Dans ce travail, nous nous concentrons sur le développement d'un modèle de communication à plusieurs niveaux d'abstraction pour la plate-forme. Un des objectifs important est le raffinement des communications qui devra se faire le plus possible de façon transparente.

L'avantage majeur de SPACE (le nom de notre plate-forme) est sans doute ses niveaux d'abstraction, qui permettent au concepteur de choisir le niveau de détail voulu pour ses simulations. L'inconvénient majeur est le fait qu'il n'y ait pas pour l'instant d'implémentation RTL équivalente au modèle raffiné.

ABSTRACT

At the system level, embedded application design can be started initially by creating a functional model, where one expresses the desired functionality of an application in terms of modules, ports, interfaces, process and channels.

An actual ongoing project inside the microelectronics research group at École Polytechnique de Montréal has for goal the creation of a SystemC platform consisting of a base architecture surrounded by helping tools for digital design conception. SystemC is a language based on C++ that makes it possible to model at a high abstraction level and to simulate hardware and software systems. The final objective of our methodology is to make it possible to the system designers to fully benefit from the platform to be able to develop their applications, without having to modify the operating system or the general components of the platform. Our tool helps in the step called module partitioning; i.e. when we decide which modules will be hardware and others software.

This work enphases on the development of a multi-level communication model for the platform. Communication refinement will be performed as much as transparent as possible for the designer.

The major advantage of SPACE (our platform's name) is undoubtedly its levels of abstraction, which make it possible for the designer to choose the right level of wanted details for his simulations. The main disadvantage is the fact that for the moment there is no RTL equivalent implementation of the refined model.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

LISTE DES NOTATIONS ET DES SYMBOLES

Les termes techniques et acronymes suivants seront utilisés dans ce document. Il s'agit de termes techniques propre au domaine ou encore des définitions propres au projet.

ACK Acknowledge, en français accusé de réception.

AMBA Advanced Microcontroller Bus Architecture, une architecture de bus commerciale.

API SystemC L'acronyme API remplace l'expression anglophone Application Programmable Interface. L'API SystemC est donc l'interface SystemC disponible au concepteur pour modéliser son application. Dans notre plate-forme, nous employons ce terme plus spécifiquement pour représenter la coucher logicielle qui fait le pont entre les modules usagers logiciels et le système d'exploitation.

ARM Advanced RISC Machine, un processeur embarqué.

ASIC Application-specific Integrated Circuit, en français circuit intégré dédié à une application.

C++ Langage de programmation orienté objet pour le logiciel.

DMA Direct Memory Access.

État bloqué On dit qu'un processus est bloqué lorsque son exécution est suspendue et que l'ordonnanceur attend qu'un événement se produise pour le remettre en exécution.

FIFO First-In-First-Out, en français premier entré premier sorti. Par contre, l'acronyme PEPS n'est pas utilisé; on dit souvent "file" ou "queue" à la place.

FIQ Fast Interrupt Request. Il s'agit d'une seconde broche d'entrée pour les interruptions sur un processeur. On peut utiliser ce port pour les interruptions qui doivent être traitées très rapidement.

HAL L'acronyme HAL désigne Hardware Abstraction Layer. Il s'agit du code du système d'exploitation qui interagit avec les périphériques matériels. Cette partie du système d'exploitation est aussi appelée le port et est dépendante de l'architecture utilisée.

ID Tous les modules de l'usager ont un identificateur unique. Cet identificateur est un nombre entier non signé de 32 bits et est utilisé pour identifier les modules émetteurs et récepteurs lors d'une communication. Nous allons utiliser le terme ID comme contraction du mot identificateur.

Instance La définition d'un module en SystemC est unique ; cependant il est possible de créer plusieurs objets distincts à partir d'une même définition, tout comme en C++ nous disons que les objets sont des instances de classes.

Interblocage En anglais, Deadlock. Situation connue en logiciel où plusieurs processus sont en attente d'évènements, de la part des autres processus, qui ne peuvent plus survenir étant donné l'état bloqué.

ISS Instruction Set Simulator. Il s'agit d'un programme qui simule l'exécution d'un processeur. Dans notre projet, le processeur cible choisi est le ARM7, et donc nous avons l'ISS de ce processeur intégré dans un SC_MODULE de SystemC, ce qui fait de l'ISS un composant de la plate-forme.

ISR Interrupt Service Routine. En français on dit routine d'interruption. C'est le code logiciel qui s'exécute lorsqu'une interruption survient.

IRQ Interrupt Request. Requête d'interruption.

Modules Entités SC_MODULE de SystemC conçus par l'utilisateur de la plate-forme pour modéliser à haut niveau son application. Les modules sont des

maîtres et peuvent communiquer entre eux en s'envoyant des messages et ils peuvent lire et écrire dans les périphériques.

Nature des modules Lorsqu'il est question de la nature des modules, nous faisons référence à la partie à laquelle le module appartient, c'est-à-dire la partie matérielle ou la partie logicielle.

OCP Open Core Protocol, un protocole standard pour la communication dans les systèmes intégrés.

OO Orienté objet, traduit de l'anglais (Object-Oriented).

Partitionnement Effectuer le partitionnement d'une application consiste à décider quels modules seront implantés en logiciel ou en matériel. Avec SPACE, il est possible à tout moment de changer le partitionnement, de recompiler le code SystemC pour obtenir un autre choix de partitionnement.

Périphériques Tous les périphériques essentiels sont fournis avec la plate-forme. Ces entités possèdent une plage d'adresse via laquelle on peut les accéder. Les principaux périphériques inclus sont les mémoires, le gestionnaire d'interruption et la minuterie pour l'horloge temps réel. Ce sont des esclaves qui ne peuvent pas initier de transactions, ils ne font que répondre aux requêtes de lecture ou d'écriture initiées par les maîtres.

Plate-forme Architecture de base modulaire configurable fournie avec notre méthodologie de conception. Le terme plate-forme désigne aussi les instances architecturales composées de modules et de périphériques propres à chaque application.

Préemption La préemption dans un système d'exploitation consiste en la suspension de l'exécution de la tâche courante par une autre tâche, causé par un événement. La préemption est effectuée sans l'intervention des tâches elles-mêmes, c'est l'ordonnanceur du noyau du système qui effectue le changement de contexte. Parfois, le mot préemption est aussi utilisé pour le cas où les

tâches se suspendent elles-mêmes et rendent la main volontairement, pour laisser l'exécution à une autre tâche. C'est ce mécanisme qui est utilisé dans le simulateur de SystemC. Dans ce document, le mot préemption sera donc employé pour désigner une cause de changement de tâche (la cause d'un changement de contexte).

RAM Random Access Memory, une mémoire à accès aléatoire. On dit de la mémoire vive qu'elle est une mémoire à accès aléatoire parce qu'elle autorise l'écriture ou la lecture des données selon leurs adresses.

RISC Reduced Instruction set Computer, ordinateur à jeu d'instruction réduit.

ROM Read-Only Memory.

RTOS Il s'agit de Real Time Operating System, en français Système d'Exploitation Temps Réel.

SoC System-On-Chip, ou System-on-a-Chip, en français, système sur puce. Un système sur puce est un système électronique complexe qui comprend habituellement un ou des processeurs, de la mémoire, un ou des réseaux d'interconnexions plus ou moins complexes, des périphériques d'entrée/sortie et souvent des parties qui ont été conçus spécifiquement pour une application particulière (ASIC), le tout sur une seule puce.

SOCP Signifie SystemC OCP.

SPACE SystemC Partitioning of Architectures for Co-design of Embedded systems, en français Outil de Partitionnement d'Architectures en SystemC pour le Codesign des Systèmes Embarqués. Il s'agit du nom de notre outil et de sa méthodologie.

SystemC Langage de modélisation au niveau système, basé sur le C++.

Tâche Une tâche est un processus en boucle qui exécute du code usager. La notion de tâche peut être attribuée au contexte SystemC ou au contexte du système d'exploitation : dans SystemC, une tâche est un processus SC_THREAD ou

SC_CTHREAD et dans le cas du système d'exploitation, une tâche est un processus du RTOS.

UTF/TF Avec SystemC il est facile de coder à plusieurs niveaux d'abstraction. Il est également facile de mélanger les niveaux. À haut niveau, nous avons le choix d'inclure les notions de temps dans le code des processus. Nous parlons alors de code SystemC fonctionnel minuté ou non (en anglais, Timed Functional et Untimed Fonctional).

VCI Virtual Component Interface, un protocole de communication.

VHDL Very High Speed Integrated Circuit Description Language ou langage de description des circuits intégrés à très haute vitesse.

VSIA Virtual Socket Interface Alliance.

LISTE DES ANNEXES

INTRODUCTION

Les systèmes embarqués sont de plus en plus présents dans notre vie quotidienne et malgré l'augmentation des coûts de production des circuits intégrés, plusieurs de ces bijoux technologiques sont offerts à des prix accessibles à tous. Bien que la concurrence agressive soit au profit des utilisateurs, elle rend la vie difficile aux manufacturiers qui doivent investir des sommes d'argent énormes pour suivre la vague technologique et assurer leur survie. Par exemple, les frais non récurrents de fabrication pour un circuit intégré numérique complexe a été multiplié par un facteur de 10 en seulement trois générations de technologies du silicium, soit 10 à 100 millions de dollars pour la technologie 0,13 micromètres [41]. Nous en sommes rendus à un point où il est théoriquement faisable d'intégrer par exemple plus de 1000 microprocesseurs à usage général sur la même puce. L'utilisation de processeurs sur puce nous permet de profiter de la flexibilité qu'offre le logiciel. On choisit habituellement d'implémenter des fonctionnalités d'un système en matériel lorsque celles-ci nécessitent une haute vitesse d'exécution ; souvent même, une implémentation matérielle de cette fonctionnalité consomme moins d'énergie qu'une version logicielle exécutée par un processeur [22].

Il est clair que des changements importants tant au niveau des méthodes de conception que des méthodes de fabrication des systèmes sur puce s'imposent. Une stratégie importante à adopter au niveau de la conception est celle de l'abstraction de la spécification. Cela implique le rehaussement du niveau d'abstraction avec lequel les concepteurs doivent travailler, la réutilisation de blocs préconçus et de bancs d'essais, l'utilisation des protocoles normalisés et la simulation de la spécification fonctionnelle exécutable.

Dans cet ordre d'idée, la version 2.0 de la bibliothèque SystemC [35] prend de plus

en plus de popularité en ce moment. Basé sur le langage C++, SystemC n'est pas une méthodologie de conception en soi, mais son utilisation suggère divers niveaux d'abstraction qui sont utiles pour établir un modèle fonctionnel ou un prototype au début d'un cycle de développement. Malgré le fait que nous pouvons qualifier SystemC de langage au niveau système, plusieurs lacunes concernant la modélisation de la partie logicielle d'une application subsistent. Les détails seront vus au chapitre 1.

Nous avons créé une plate-forme en SystemC, nommée SPACE (*SystemC Partitioning of Architectures for the Co-design of Embedded systems*) [10, 4] qui veut répondre à ce problème en incluant un système d'exploitation temps réel dans une simulation en SystemC. Notre principale contribution réside dans la méthodologie qui vient avec notre plate-forme. Nous proposons deux niveaux d'abstraction différents : un qui permet d'être indépendant de toute architecture et un autre, constitué cette fois d'une partie logicielle et d'une partie matérielle, qui permet de fournir une simulation plus détaillée et qui donne des informations au concepteur, le guidant ainsi dans ses choix architecturaux. L'usager peut prendre ses modules conçus et vérifiés au premier niveau (nommé UTF) et les connecter à la plate-forme (niveau TF) sans modifications majeures. L'avantage de ces deux niveaux est qu'il est possible de revenir en arrière sur un choix de partitionnement logiciel/matériel et d'en essayer plusieurs. Néanmoins, notre première implémentation comporte plusieurs faiblesses étant donné qu'il ne s'agit que d'un premier prototype.

Ce mémoire s'intéressera à la mise en oeuvre des moyens de communication sur cette plate-forme. Concrètement, le travail réalisé consiste à la définition de deux niveaux d'abstraction et des règles invariantes pour un utilisateur qui en découlent, à la conception et l'implémentation des interfaces pour la communication, des canaux de communication, des périphériques de la plate-forme et des mécanismes de communication logicielle/matérielle.

On attribuera ainsi les deux principales contributions suivantes au travail réalisé dans le cadre de ce mémoire :

- Une plate-forme qui consiste en une série de périphériques réutilisables et de règles d'implémentation, laissant l'usager créer des architectures adaptées à ses applications ;
- Des interfaces de communication unifiées qui peuvent être utilisées pour différents niveaux d'abstraction et pour des modules logiciels et matériels, simplifiant le raffinement et l'exploration architecturale d'une application en SystemC.

Mathieu Rondonneau et Jérôme Chevalier ont contribué au projet SPACE, respectivement à la conception et au développement de la partie logicielle [42] et à l'intégration d'un simulateur de processeur dans la simulation SystemC [9].

Au premier chapitre, nous rappellerons l'importance de la spécification unifiée à un seul langage, le C++ dans notre cas, et nous verrons les concepts de base du langage SystemC. Une brève analyse de celui-ci fera ressortir les lacunes en ce qui concerne la simulation du logiciel. Nous présenterons ensuite plusieurs niveaux d'abstraction qu'un utilisateur de SystemC peut exploiter et finalement nous verrons plusieurs approches au raffinement d'une spécification en SystemC.

Le chapitre 2 est consacré à la description de la plate-forme SPACE, notre solution au problème du partitionnement logiciel/matériel. Nous y verrons le fonctionnement et les concepts généraux, détaillerons la structure de la plate-forme et présenterons le flot de conception pour décrire son utilisation.

C'est au troisième chapitre que nous détaillerons le fonctionnement interne des canaux de communication de la plate-forme. D'abord, le sujet sera abordé avec une

vue d'ensemble. Ensuite les multiples interfaces pour permettre l'échange de messages entre les différents composants de l'usager et périphériques seront exposées. Nous terminerons le chapitre par la description détaillée d'abord des composants qui participent à la communication matérielle puis finalement, des blocs conçus pour assurer le support logiciel dans SPACE.

Le dernier chapitre discute des résultats de performance de SPACE en comparaison avec d'autres modèles de communication existants et aussi à travers une série de tests paramétriques qui ont pour but de montrer les forces et les faiblesses de notre implémentation. Un exemple d'utilisation de la plate-forme en quatre versions y est également présenté. Ensuite, nous expliquons les variantes dans l'implémentation qui pourraient améliorer les performances au niveau de la communication matérielle.

Nous concluons ce mémoire en effectuant un survol des améliorations et ajouts qui pourraient être portés sur la plate-forme pour en faire un outil de codesign plus complet.

CHAPITRE 1

REVUE DE LITTÉRATURE ET PRÉALABLES

1.1 Modélisation de systèmes et techniques orientées objet

La programmation orientée objet (OO) fait partie de la vague de changement au niveau des méthodologies de conception dans le monde des systèmes intégrés (ou systèmes sur puce, ou encore systèmes embarqués). L'OO s'intègre aux méthodes existantes de spécification à plus haut niveau. Ces techniques qui rehaussent le niveau d'abstraction peuvent nous permettre de spécifier la fonctionnalité d'une application sans définir à prime abord ce qui sera implémenté en logiciel et en matériel. L'orienté objet ne s'applique pas vraiment pour ce qui est de la spécification non fonctionnelle, à savoir les contraintes de temps réel (temps de réponse), les fréquences minimales d'exécution, la puissance maximale d'énergie à dissiper ou consommer, la durée de vie souhaitée d'une pile, la surface consommée sur une puce électronique, etc.

La spécification fonctionnelle est souvent exprimée par un ensemble de processus ou tâches communicantes [22, 12]. Les réseaux de processus de Kahn (Kahn Process Networks) [25] ont l'avantage de posséder la propriété d'ordancement déterministe lors des multiples exécutions en simulation. Rappelons-le, les réseaux de processus de Kahn sont des processus concurrents, munis de canaux de communication unidirectionnels de type FIFO de tailles infinies. Chaque processus s'exécute séquentiellement et de façon privée par rapport aux autres processus et la synchronisation est implicite à la communication. Les lectures sur les canaux sont bloquantes, c'est-à-dire qu'un processus attend d'avoir assez de données disponibles

pour compléter l'opération de lecture dans le cas où le canal serait vide de message. Les écritures sont non bloquantes au sens où les FIFO sont infinis et donc les processus n'ont pas à se soucier si les canaux sont pleins. De plus, il n'y a pas d'accusés de réception sur les écritures. L'ordonancement des processus de Kahn est en effet déterministe, car l'ordre des échanges de données dépend des entrées et non de l'ordre d'exécution des processus. Par contre, on ne peut pas modéliser un système réactif complet, avec interaction de l'usager par exemple. De plus, d'un point de vue pratique, il est impossible d'implémenter un système avec des canaux de longueur infinie.

Néanmoins, la spécification fonctionnelle d'un système numérique est toujours faite en pratique avec un langage qui permet de définir les mêmes concepts. La spécification est divisée en modules ou entités qui contiennent des éléments d'exécution actifs ou réactifs, sensibles à des événements ou à des signaux externes et qui communiquent entre eux par des canaux dédiés ou des canaux implicites. Tel est le cas de Verilog [23], VHDL [24], Cynlib [15, 16], Syslib [14] et bien d'autres.

SystemC [35, 45, 36, 20, 37] est un de ceux là et probablement le plus populaire en ce moment. SystemC offre l'avantage d'être basé sur le langage C++ [13] et donc met à profit les concepts orientés objet pour la spécification au niveau système. De plus, dans un contexte de conception des systèmes matériels/logiciels, le fait de spécifier la partie logicielle du système avec le C++ rend instinctivement cette partie du système facile à raffiner.

1.2 SystemC

Nos travaux utilisent SystemC et il est important de le décrire, ne serait-ce que brièvement.

1.2.1 Composants de base pour la modélisation

Avec sa version 2.0.1, SystemC fournit les éléments essentiels à la modélisation au niveau système : les modules, les ports avec fonctionnalités étendues, les événements, la sensibilité dynamique, les interfaces et les canaux définis par l'utilisateur, et bien plus. Voici une brève description des caractéristiques intéressantes.

Modules (SC_MODULE) Les modules sont les entités ou composants de base pour décrire de façon regroupée et hiérarchique un ensemble de fonctionnalités communes. Les modules sont des éléments structurels.

Processus (SC_THREAD, SC_CTHREAD, SC_METHOD) Les processus sont des éléments fonctionnels. Ils peuvent être perpétuels (`SC_THREAD`), perpétuels synchrones (`SC_CTHREAD`) ou momentanés (`SC_METHOD`). Tous les processus décrivent des exécutions séquentielles. Les processus perpétuels peuvent être interrompus et réactivés, tandis que les `SC_METHOD` s'activent, s'exécutent et se terminent sans interruption.

Interfaces (sc_interface) Une interface définit de façon abstraite un sous-ensemble de méthodes (ou fonctions) qui peuvent être utilisées par les modules pour communiquer avec les autres modules, en utilisant les canaux.

Canaux définis par l'usager (sc_channel) Les canaux sont des entités de type `SC_MODULE` spéciaux, car ils implémentent la communication entre les modules, en fournissant une implémentation aux interfaces.

Port (sc_port) Un port donne à un module un accès externe, via une interface. Pour connecter un module à un canal, on doit utiliser un port compatible avec le canal. La compatibilité assure que le port utilise la même interface que celle qui est implémentée dans le canal.

Synchronisation (wait()) La fonction `wait()` suspend l'exécution d'un processus perpétuel de SystemC qui appelle cette fonction. L'argument passé à la

fonction dépend du type de processus perpétuel et sert à déterminer la condition qui dicte quand le processus pourra reprendre son exécution (notation temporelle ou événement prédéterminé).

Événements (sc_event) Les événement sont utilisés pour la synchronisation interprocessus dans un même module. Ils sont utilisés pour suspendre et activer les processus de SystemC.

1.2.2 Exemples

À l'aide de deux exemples très simples, nous allons présenter un résumé des fonctionnalités utilisées dans le cadre de notre projet. L'annexe IV rassemble le code complet des deux exemples. La figure 1.1, inspirée de [20], montre une notation graphique qui est souvent utilisée pour SystemC.

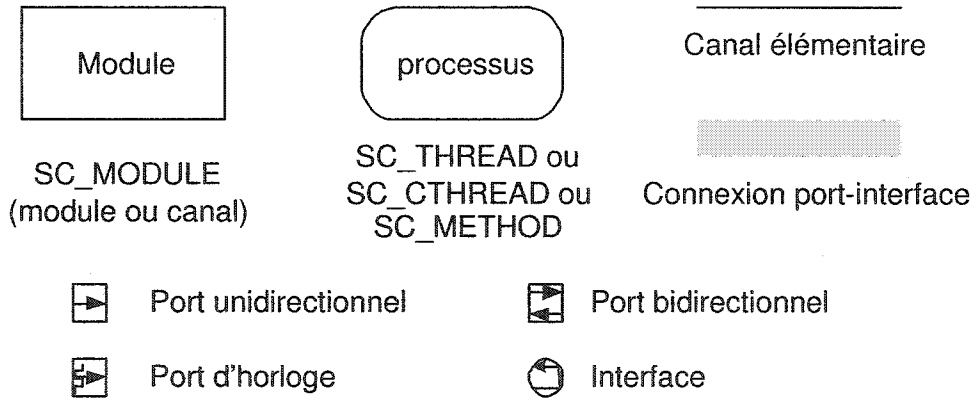

FIGURE 1.1 Notation graphique pour SystemC

La première version de l'exemple, celle de la figure 1.2, utilise deux modules qui contiennent chacun un processus de type `SC_THREAD`. Le processus de `prod1` envoie des données au processus `cons1`, par le biais d'une queue de message de type

sc_fifo. Le code du module prod1 est le suivant :

```
SC_MODULE(prod)
{
    sc_fifo_out<int> port;
    void thread(void) {
        for (int i = 1; i<=10; i++) {
            port->write( i*100 );
        }
    }
    SC_CTOR(prod) {
        SC_THREAD(thread);
    }
};
```

Nous pouvons voir que le code est en fait du C++ avec des macros et des objects spéciaux. La macro SC_MODULE permet de construire une classe qui hérite de type la classe de base sc_module. Le constructeur de la classe est également créé à partir d'une macro : SC_CTOR. Dans le constructeur, on déclare la méthode membre thread() comme étant un processus de SystemC de type SC_THREAD. Le port du module est déclaré comme un aggrégat de la classe prod et la méthode write() est utilisée pour écrire une donnée dans le fifo.

Les deux objets prod1 et cons1 sont des instances des classes prod et cons, construit dans le module de plus haut niveau que nous avons appellé top1. Cet objet est simplement le module qui contient tous les autres composants pour la simulation. Le code de la classe top est le suivant :

```
SC_MODULE(top)
```

```
{
    prod* prod1;
    cons* cons1;
    sc_fifo<int>* fifo1;

    SC_CTOR(top) {
        prod1 = new prod("prod1");
        cons1 = new cons("cons1");
        fifo1 = new sc_fifo<int>(1);

        prod1->port(*fifo1);
        cons1->port(*fifo1);
    }
};
```

On peut remarquer la façon dont on connecte un port à un canal : il suffit d'utiliser la notation module-¿port(*canal). Afin de simuler l'exemple, il suffit de construire un objet de la classe top et de lancer la simulation en utilisant la fonction `sc_start()` :

```
int sc_main(int argc, char**argv)
{
    top top1("top1");
    sc_start(-1);
    return 0;
}
```

Le deuxième exemple est une variante du premier pour expliquer d'autres constructions de SystemC. Il faut se référer à la figure 1.3. Au lieu d'utiliser un canal de type

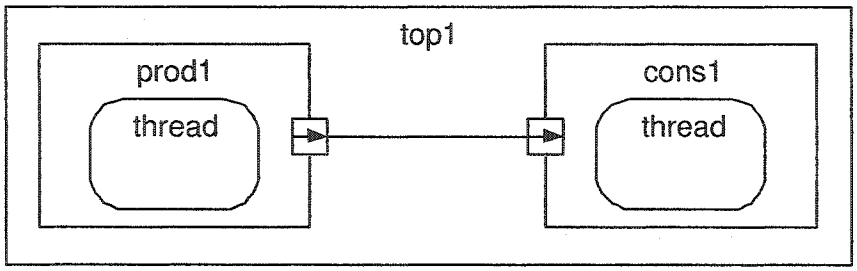

FIGURE 1.2 Exemple SystemC #1

`sc_fifo`, nous avons construit un canal sur mesure. Pour ce faire, il faut d'abord établir l'interface, c'est-à-dire les méthodes permises pour accéder au canal :

```
class mychannel_if: public sc_interface
{
public:
    virtual void mychannel_read(int& value) = 0;
    virtual void mychannel_write(int value) = 0;
};
```

Ensuite, nous pouvons créer le canal, qui fournit une implémentation aux méthodes de l'interface. Le canal est un module, il doit hériter de la classe de base `sc_module` (ou `sc_channel`, qui est la même) :

```
class mychannel: public sc_channel, public mychannel_if
{
    // [...]
    sc_event write_event;
    sc_event read_event;
    virtual void mychannel_read(int& value) {
```

```
        if (empty)
            wait(write_event);
        value = internal;
        empty = true;
        read_event.notify();
    }
    virtual void mychannel_write(int value) {
        if (!empty)
            wait(read_event);
        internal = value;
        empty = false;
        write_event.notify();
    }
};
```

Nous pouvons remarquer l'utilisation des événements `read_event` et `write_event` ainsi que de la fonction d'attente `wait()` pour synchroniser les processus à même le canal.

Pour cet exemple, nous avons remplacé les processus `SC_THREAD` par des processus `SC_CTHREAD`. Par conséquent, les modules doivent avoir un port d'entrée pour l'horloge, qui est désormais nécessaire. Voici à quoi ressemble le code du module `prod2` pour cet exemple :

```
class prod : public sc_module
{
public:
    sc_port<mychannel_if> port;
    sc_in_clk clock;
```

```
    SC_HAS_PROCESS(prod);

    void cthread(void) {
        for (int i = 1; i<=10; i++) {
            port->mychannel_write( i*100 );
        }
    }
    prod(sc_module_name name_) : sc_module(name_) {
        SC_CTHREAD(cthread, clock.pos());
    }
};
```

La macro `SC_HAS_PROCESS` indique au simulateur que la classe `prod` contient des processus. Cela nous permet d'utiliser un constructeur avec d'autres arguments que ce qu'impose la macro `SC_CTOR`, tout en pouvant déclarer des processus.

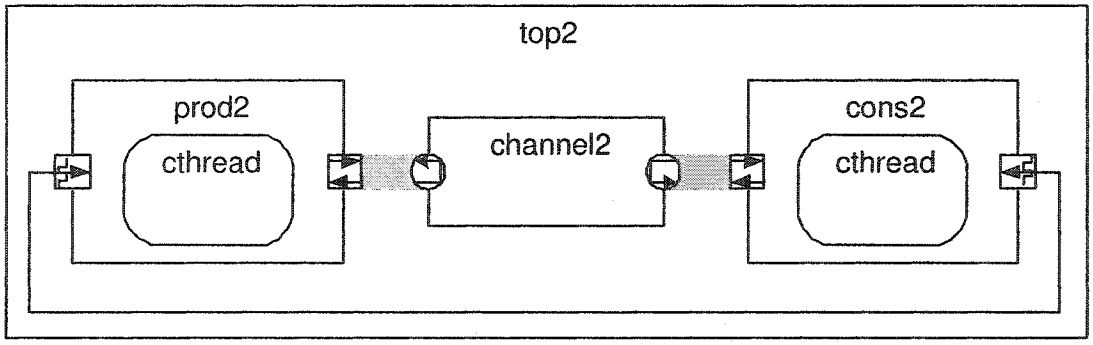

FIGURE 1.3 Exemple SystemC #2

Il existe une documentation relativement complète [38, 35, 45, 36, 20, 7, 37] pour le lecteur intéressé à en connaître davantage sur SystemC.

1.2.3 Lacunes

Malgré le fait que SystemC 2.0 se veut un outil de modélisation au niveau système, il n'en demeure pas moins qu'il est plus approprié pour la modélisation du matériel. À haut niveau, SystemC permet de séparer l'application en plusieurs modules hiérarchiques qui contiennent des processus décrits dans le même langage pour le matériel et pour le logiciel. Par contre, tous les dispositifs sont en place pour permettre le raffinement du matériel, mais pas pour le logiciel. Le raffinement est par conséquent freiné.

Le problème vient du fait que l'ordonnanceur est le même pour le logiciel et pour le matériel lors des simulations [30, 7]. On ne peut donc pas modéliser les propriétés du logiciel. La concurrence n'est pas modélisée, ni l'ordonnancement préemptif, ni le support des priorités. Il est donc impossible de gérer les systèmes avec des contraintes de temps réel dures (i.e. des contraintes critiques de temps). Pourtant, les systèmes temps réel sont beaucoup utilisés dans les systèmes sur puce, notamment dans les systèmes réactifs. Pouvoir les modéliser en tant que parties d'un système complet est un atout stratégique évident pour le codesign [6]. Ces fonctionnalités manquantes dans SystemC devraient être comblées par la prochaine version, pour laquelle un support pour la modélisation des systèmes d'exploitation temps réel, offrant ainsi un environnement pour la conception logicielle des systèmes embarqués, sont attendues [7].

Pour l'instant, plusieurs solutions ont été suggérées, notamment l'utilisation de modèles pour les RTOS [18] ou encore l'intégration de ISS dans SystemC [3, 38]. Ceci implique qu'il faut trouver des techniques pour effectuer de la simulation conjointe du matériel et du logiciel pour contrer les lacunes.

Pour le lecteur intéressé, l'annexe I traite de la co-simulation plus en détails.

1.3 Modélisation architecturale et raffinements

En plus de pouvoir modéliser et simuler des systèmes logiciels/matériels, nous devons pouvoir raffiner les spécifications. Cela permet au concepteur d'ajouter des détails à son modèle, tant au niveau structurel, fonctionnel que temporel.

1.3.1 Niveaux d'abstraction avec SystemC 2.0

SystemC à lui seul supporte une modélisation avec des sous parties à différents niveaux d'abstraction. On peut commencer à spécifier une application au niveau fonctionnel, ensuite ajouter des notions de temps et ainsi tendre petit à petit vers le niveau RTL (ou BCA) [44]. Le raffinement des communications peut s'effectuer en partant de canaux abstraits (c'est-à-dire de faire une implémentation purement fonctionnelle des interfaces) et tendre vers un modèle qui intègre le canal de communication complet, tant du point de vue de la structure que de la précision au niveau des cycles, en respectant un protocole par exemple. Le raffinement en SystemC permet aussi de partir des types de données implicites du C++ (`int`, `float`, `char`) et de les spécialiser vers des types de données plus spécifiques (nombres à virgule fixe comme `sc_fix`, vecteurs de bits de longueur définie comme `sc_lv`).

Kurt Schwartz [44] propose les étapes suivantes pour le raffinement :

1. Raffinement du canal de communication ;
2. Insertion d'adaptateurs, pour permettre de garder les modules à haut niveau et le canal à un niveau plus raffiné, c'est-à-dire d'une part de détailler au niveau des signaux et des broches la communication qui était définie auparavant par les méthodes de l'interface ;
3. Validation du système ;

4. Raffinement des interfaces au niveau des modules, en supprimant les adaptateurs et en insérant à même les processus des modules les détails de la communication ;
5. Validation du système.

Dans [39], on y présente les avantages de la modélisation au niveau transactionnel, ou TLM (Transaction Level Modeling). Une transaction est un échange de données ou d'événements entre deux composants. Au niveau TLM, il est plus facile de modifier le partitionnement logiciel/matériel qu'au niveau RTL. D'autres avantages sont attribuables au TLM :

- Développement rapide de modèles ou prototypes fonctionnels ;
- Exécution de la spécification ;
- Simulation rapide ;
- Possibilité de faire une première estimation de performance ;
- Mécanismes (métriques) pour supporter l'exploration architecturale ;
- Réutilisation des composants d'un système à un autre.

Dans [31, 39], on y présente aussi trois niveaux différents d'abstraction notables :

1. Le niveau fonctionnel, qui est indépendant de l'architecture, est composé de constructions comportementales ;
2. Le niveau architecture, dépendant de l'implémentation, où on peut distinguer les logiciel et le matériel, mais où on conserve un niveau d'abstraction assez élevé ;
3. Le niveau microarchitecture, précis au niveau des cycles avec les broches et les signaux.

Le niveau TLM correspond donc au deuxième niveau présenté. Il est à noter que

l'auteur ne présente pas toutefois comment passer d'un niveau à l'autre. À ce sujet, le chapitre 3 de [7] discute assez en détails du raffinement avec SystemC. On y présente un flot de conception comparable à [39]. Voici un résumé des étapes du raffinement :

UTF (Untimed functional) Ce niveau permet de définir les interfaces et de donner une première implémentation fonctionnelle des canaux de communication.

TF (Timed Functional) Le raffinement débute à ce niveau, où on commence par ajouter des délais dans les processus. Pour ce faire, il faut transformer les `SC_METHOD` en `SC_THREAD` ou `SC_CTHREAD`. Il faut parfois modifier les algorithmes à l'intérieur des processus pour pouvoir mieux les évaluer. On peut aussi raffiner les types de données du C++ vers des types de données propres à SystemC. Au niveau des communications, on peut essayer d'évaluer grossièrement le temps d'attente et insérer des délais correspondants, au niveau des fonctions d'entrées/sorties.

BCA (Bus-Cycle Accurate) À ce niveau on raffine maintenant le modèle de communication. On passe d'une synchronisation par événements en communication synchrone sur le front de l'horloge. Il peut être alors pratique d'insérer des adaptateurs pour conserver la même interface au niveau des modules, mais avec un raffinement au niveau du canal, un peu comme le propose [39].

PA (Pin Accurate) Comme son nom le sous-entend, à ce niveau on décompose les interfaces en signaux et données élémentaires. Les comportements sont ainsi décomposés en contrôleurs et chemins de données, en vue de la synthèse comportementale et logique. La partie contrôleur est implémentée comme une machine à état et souvent il faut ajouter des ports pour la synchronisation entre les parties de contrôle et de calcul.

CA (Cycle-Accurate) Ce niveau résulte de la fusion des modifications entre les

niveaux BCA et PA. En d'autres mots, les adaptateurs qui servaient de pont entre les modules et le canal sont supprimés et tout le système résultant est précis au niveau des cycles et au niveau de la structure.

Une autre approche de raffinement a été proposée, mais cette fois, de façon étroitement liée avec un protocole qu'ils nomment SOCP [21] (pour SystemC OCP). OCP [33] est un protocole de communication standard dans l'industrie, que l'on peut comparer avec VCI, de VSIA [34, 11, 22].

Le raffinement proposé se divise en quatre niveaux distincts avec SystemC. Encore une fois, par contre, tout l'aspect logiciel est négligé. Le raffinement matériel se compare beaucoup à notre approche, c'est pourquoi il est important ici de bien décrire les niveaux un par un. Le tableau 1.1, tiré de [21], présente les couches (ou niveaux) ainsi que les détails qui y sont abstraits.

TABLEAU 1.1 Niveaux d'abstraction (couches) pour SOCP

Acronyme	**Couches**	**Ce qui est abstrait**
L3	Couche message	Partage des ressources, temps
L2	Couche transaction	Horloges, protocoles de communication
L1	Couche transfert	Fils, registres
L0	Couche RTL	Portes logiques, délais dans les fils et portes

L0 Il s'agit du fameux niveau RTL bien connu. À ce niveau, la description comporte assez de détails pour être précise au niveau des broches, des bits, des cycles et du contenu des registres. Il s'agit d'un code final prêt pour le processus de synthèse (VHDL, Verilog ou SystemC synthétisable). Il ne faut pas exclure qu'il faudra optimiser le circuit résultant, faire le placement/routage, faire la distribution des horloges, de l'alimentation, etc.

L1 Le niveau 1 est précis au niveau des cycles (Cycle-True), c'est-à-dire que le

système a le même comportement que le code RTL. La précision dans les communications permet de respecter le protocole choisi et les entrées/sorties. Cependant, l'utilisation des interfaces limite la précision des données échangées au niveau des octets ou mots. Les avantages sont que les interconnexions sont plus simples et la simulation plus rapide, étant donné qu'on supprime des signaux et qu'on les remplace par des événements.

L2 Le niveau 2 fait abstraction des signaux d'horloge. La notion de temps peut toujours exister, mais il n'y a pas de précision au niveau des cycles. La synchronisation est assurée par des événements avec des annotations temporelles, ce qui permet de paramétrer les canaux de communication pour faire des ajustements assez précis au niveau des performances. Ce niveau est idéal pour l'estimation de performance en vue du partitionnement logiciel/matériel.

L3 Le plus haut niveau est fonctionnel sans notion de temps. On y trouve comme avantage la facilité du partitionnement fonctionnel. Ce niveau est abstrait de toute architecture, donc les canaux de communication implémentent le minimum, à la manière du point à point.

1.3.2 Abstraction de protocoles

Comme nous venons de le voir, le raffinement peut être lié à un protocole de communication. Cela permet entre autre de s'assurer que le système pourra être raffiné peu à peu pour tendre vers un protocole existant, standard, que les ingénieurs pourront réutiliser et pour lequel le processus de synthèse ne posera pas de problèmes majeurs.

En ayant un protocole défini, on peut aussi définir des interfaces SystemC unifiées et il est alors possible d'utiliser les mêmes interfaces à plusieurs niveaux d'abstraction. Cela permet de mixer les niveaux ou encore de raffiner le canal progressivement sans

avoir à toucher au code des modules, en utilisant des adaptateurs. Si les niveaux d'abstractions sont eux aussi bien définis, les adaptateurs peuvent être conçus une seule fois et réutilisés dans plusieurs designs.

Nous avons expliqué le raffinement en quatre couches avec SOCP. Une plate-forme nommée StepNP [41] utilise aussi un protocole avec le même nom. SOCP signifie encore "SystemC OCP", pour la même raison qu'ils ont abstrait le niveau d'abstraction de OCP. Par contre, le premier objectif de leur plate-forme n'est pas le raffinement, ils fournissent plutôt une série d'outils et de composants sous forme de plate-forme plus ou moins générique pour les applications réseaux. L'emphase est mise sur l'intégration d'un certain nombre de processeurs RISC dans une même architecture. Les interconnexions avec SOCP sont de niveau TLM et plusieurs modèles sont supportés.

Parfois, on peut définir un protocole à haut niveau (TLM) sans pour autant être basé sur un protocole de bas niveau existant. Un exemple est Simple Bus, bien décrit dans [20]. Ce bus transactionnel est synchrone sur l'horloge et utilise les deux fronts (montants et descendants) pour faciliter l'arbitrage. Il est à remarquer que cela n'est qu'une technique de modélisation ; le système final raffiné pourrait être doté d'une seule horloge qui active la logique sur le front montant seulement.

Simple Bus comporte trois jeux d'interfaces, certaines pour les blocs maîtres, d'autres pour les blocs esclaves et une interface pour l'arbitre du bus.

Pour les esclaves (par exemple les mémoires et les périphériques) :

- Interface normale ;
- Interface directe.

Pour les maîtres (par exemple les processeurs ou DMA) :

- Interface bloquante ;
- Interface non bloquante ;
- Interface directe.

On utilise l'interface de l'arbitre pour se renseigner sur quel maître obtiendra l'accès au canal. Les interfaces maîtres et esclaves directes servent à ignorer le protocole, à titre informatif ou pour des fins de déverminage. Il est possible de verrouiller le bus pour plusieurs cycles par un maître, pour simuler un transfert en rafale (burst). Au niveau des esclaves, il est possible d'ajouter des cycles de latence pour simuler un délai de traitement.

Le problème avec Simple Bus, c'est qu'il n'existe pas de modèle raffiné équivalent au niveau RTL. Une bonne approche serait de modifier le modèle pour modéliser la structure et les caractéristiques temporelles d'un protocole de bus semblable, mais standard, comme AMBA [2, 5] par exemple.

CHAPITRE 2

PRÉSENTATION DE LA PLATE-FORME

2.1 Objectifs de la plate-forme

La plate-forme SPACE [10, 4] peut remplir plusieurs fonctions. Les objectifs sont :

1. Permettre la conception et la simulation à haut niveau de plusieurs configurations logicielles/matérielles d'une application de codesign ;
2. Fournir des composants réutilisables comme base pour les concepteurs, pour éviter de recommencer de zéro la conception pour chaque application ;
3. De permettre l'exploration architecturale afin de pouvoir changer la nature d'un module en cours d'exploration ;
4. Produire des résultats de simulation sur lesquels le concepteur pourra évaluer ses choix de partitionnement.

Une application de codesign est une application qui est composée à la fois de parties logicielles et de parties matérielles. Le codesign concerne surtout la phase de mise en fonction des deux parties simultanées, c'est-à-dire lorsque les parties logicielles et matérielles doivent interagir.

Lors de la phase de spécification de l'application, il faut effectuer un partitionnement en modules. Ceci consiste à diviser en plusieurs entités, blocs, parties, composants ou encore modules la fonctionnalité du système. Cette phase établie un partitionnement structurel. Il faut encore ensuite décider quels modules doivent

être implantés en logiciel ou en matériel. Il s'agit alors du partitionnement logiciel/matériel. On nomme configuration logicielle/matérielle ou partition chacune des possibilités architecturales issues d'un partitionnement.

Pour concevoir un modèle exécutable de la spécification d'une application de co-design, nous utilisons le langage de modélisation SystemC version 2.0. Le niveau d'abstraction choisi est élevé, ce qui nous permet au besoin d'omettre les détails architecturaux et structurels afin de rester au niveau purement fonctionnel. Avec la plate-forme SPACE, un concepteur système réutilise les composants de base que nous avons conçus. Ces composants génériques et configurables définissent une architecture de base et permettent de réduire le temps de conception. Le but premier de la plate-forme est de permettre l'exploration architecturale d'une application ou d'un modèle décrit en langage SystemC de haut niveau. La plate-forme facilite le travail du concepteur lorsque ce dernier désire essayer différentes configurations logicielles/matérielles, car à ce niveau il est possible de déplacer les modules de la partie logicielle vers la partie matérielle et inversement, avec un effort mineur et un temps minimum de la part du concepteur. Pour guider le concepteur dans son partitionnement, il lui est possible d'effectuer des simulations de son application. À partir des résultats, il peut réitérer et converger vers un partitionnement optimal.

2.2 Description sommaire de notre solution

Afin de rendre compte de ces objectifs, nous avons choisi de bâtir un environnement de simulation séparé pour la partie logicielle et pour la partie matérielle. Les deux simulations doivent par contre être cohérentes et synchronisés. Pour ce faire, nous avons décidé d'intégrer un ISS dans une plate-forme en SystemC. Cet ISS se retrouve dans un module et donc SystemC s'occupe de faire exécuter les processus matériels et l'ISS. Comme l'ISS est également lui-même un simulateur, un second

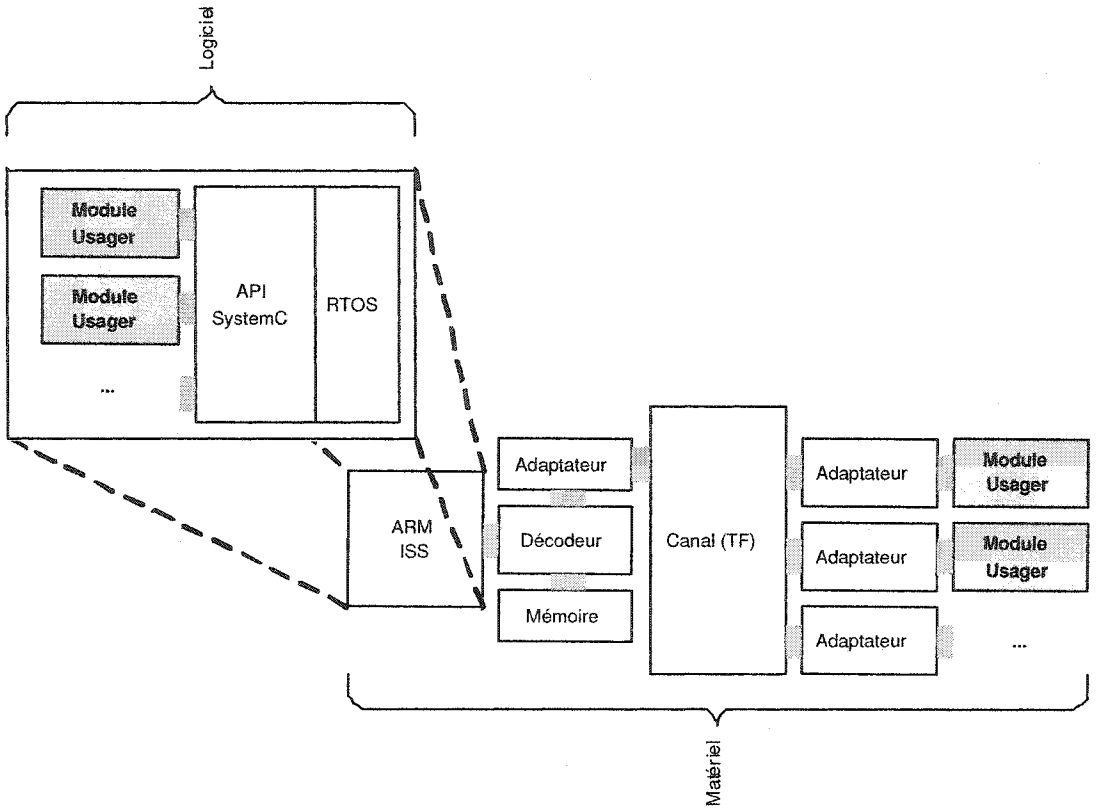

FIGURE 2.1 Schéma de principe de la plate-forme

domaine de simulation est possible comme l'illustre la figure 2.1. Pour l'instant, SPACE ne possède qu'un seul modèle d'ISS : celui du ARM7 [1]. Nous pourrions faire exécuter n'importe quel code sur cet ISS ; cependant, l'astuce de SPACE est de faire exécuter sur le processeur du code SystemC. Ce code SystemC n'est pas exécuté par le même simulateur que le code SystemC de la partie matérielle. Ainsi nous avons deux simulateurs de SystemC.

SystemC n'est pas un système d'exploitation ; ce n'est qu'une bibliothèque d'objets en C++. Par conséquent, il nous faut un système d'exploitation pour pouvoir simuler du code conçu avec SystemC. C'est ainsi que fonctionne la simulation de la partie matérielle. Ce n'est toutefois pas le cas pour la partie logicielle. Nous

avons plutôt choisi d'utiliser un système d'exploitation temps réel, MicroC/OS-II [26], muni d'une bibliothèque qui permet de fournir l'interface de programmation de SystemC sur un processeur ARM. Ainsi, au lieu d'utiliser un système d'exploitation complet (comme Windows ou Unix) avec la bibliothèque SystemC, nous avons opté pour une solution plus appropriée pour les systèmes embarqués. Au lieu d'avoir recours à un simulateur dédié pour ordonnancer les processus de SystemC, c'est l'ordonnanceur du micronoyau de MicroC/OS-II qui gère les processus de SystemC. Par exemple, lorsque le concepteur incorpore un `SC_CTHREAD` dans un de ses modules, il y a véritablement une tâche de MicroC/OS-II qui est crée et exécutée. L'interface de programmation permet d'émuler la plupart des éléments de modélisations fournis traditionnellement avec la bibliothèque SystemC.

Nous devons également assurer les communications entre les parties logicielles et matérielles, pour que les modules puissent s'envoyer des messages, peu importe leur nature.

2.3 Justifications des choix

Nous avons choisi le langage de modélisation SystemC d'une part parce qu'il s'agit présentement du langage le plus populaire pour la modélisation et la simulation de systèmes et aussi parce que le code source est ouvert. Cette deuxième propriété nous a permis de concevoir rapidement l'API SystemC en logiciel à partir du code existant de SystemC. Le choix du processeur s'est arrêté sur le ARM7. C'est un des plus utilisés dans les systèmes embarqués et nous avions le code source de son ISS, gracieuseté de GNU [19]. De plus, nombreux sont les outils de développement et les systèmes d'exploitation qui supportent le ARM. MicroC/OS-II a été choisi comme système d'exploitation principalement parce que le groupe de recherche était familier avec celui-ci. De plus, nous avons le code source complet et certains

ports ont déjà été conçus pour le ARM7.

2.4 Fonctionnement et concepts

Nous allons maintenant expliquer le fonctionnement général de la plate-forme et les concepts nécessaires pour comprendre le contexte d'interaction des canaux de communications.

2.4.1 Caractéristiques des modules de l'usager

Tout le code de l'utilisateur est en SystemC. La plate-forme SPACE est elle-même décrite en majorité en langage SystemC, à l'exception de certains composants comme le système d'exploitation et le coeur de l'ISS. Le code de l'utilisateur se retrouve seulement dans la description en SystemC des modules et dans la description des interconnexions. Les modules doivent hériter d'une classe de base définie dans SPACE. Il y a deux classes de base, dépendamment du niveau d'abstraction où on désire simuler. Chaque instance de module possède un numéro d'identification unique donné par le concepteur et un port de communication qui utilise une interface spécifique à SPACE. Certains numéros d'identification (comme le zéro) doivent être réservés pour un usage interne de la plate-forme. Ces numéros jouent le rôle des adresses pour les modules ; cependant, en ayant des numéros uniques qui définissent les modules, il est plus facile de changer la nature des modules tout en garantissant la communication. Lorsque les modules sont bougés dans la partie matérielle de l'application, il a y toujours une adresse où les modules logicielles doivent envoyer leurs messages pour garantir la communication. Néanmoins, cette transformation est faite automatiquement à l'interne à partir d'une adresse définie dans la configuration de la plate-forme et le concepteur n'a pas à s'en soucier.

2.4.2 Communication

Voici une description de l'interface de communication des modules, soit les méthodes avec leurs paramètres, accompagnées d'une brève description.

read() et write() (unsigned long MyID, unsigned long TargetID, unsigned long Priority, void* Data32, unsigned long DataLength8) : À l'aide de son port unique, le module peut appeler une fonction de lecture ou d'écriture bloquante, c'est-à-dire que la fonction retourne uniquement après que la réception du message est confirmée. Dans les deux cas, le module qui fait une requête doit s'identifier (MyID) et spécifier à quel module est adressée la requête. Le module qui fait la requête peut être vu comme le maître et celui qui répond comme l'esclave.

nb_read() et nb_write() (unsigned long MyID, unsigned long TargetID, unsigned long Priority, void* Data32, unsigned long DataLength8) : Lecture et écriture avec comme seule différence qu'elles sont non bloquantes. On peut savoir qu'une opération s'est bien terminée en analysant le code de retour des méthodes.

mem_read() et mem_write() (unsigned long MyID, unsigned long Address, unsigned long Priority, void* Data32, unsigned long DataLength8) : Ces deux fonctions (toujours bloquantes) constituent l'interface utilisée par un module pour communiquer avec les périphériques esclaves du système. Tous les périphériques sont adressables comme une mémoire.

Nous avons défini une interface unique pour permettre aux modules usager de communiquer entre eux et avec les périphériques de la plate-forme. Tous les modules usager possèdent un port de communication qui utilise cette interface, détaillée au tableau 2.1. À l'aide des méthodes membres, il est possible d'envoyer et de recevoir

des messages. La communication entre les modules fonctionne un peu comme dans un réseau. Les données sont encapsulées dans des paquets qui contiennent aussi une entête avec les numéros d'identification du module émetteur et du récepteur ainsi que la taille du message. Un tel paquet est illustré à la figure 2.2.

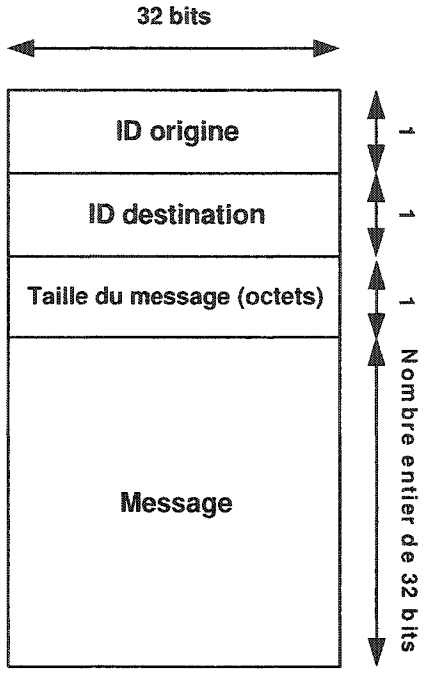

FIGURE 2.2 Schéma d'un paquet

Le concepteur a le choix d'utiliser la communication bloquante ou non bloquante. Nous désignons bloquante une communication où l'émetteur attend que le récepteur ait pris connaissance du message avant de continuer à s'exécuter. Dans le cas d'une écriture, l'émetteur attend que le récepteur ait lu le message envoyé. Pour le cas d'une lecture, la communication bloquante signifie que le module qui effectue la lecture doit attendre que le message soit émis avant de continuer à s'exécuter, à la manière d'une synchronisation par rendez-vous [46]. Cela semble bien trivial, par contre on peut toutefois vouloir utiliser la lecture non bloquante dans le cas l'où désire vérifier si un message a été envoyé et agir différemment selon le cas. L'écriture

non bloquante, quant à elle, est souvent utilisée dans le cas où un module doit envoyer simultanément des messages à plusieurs modules. Lorsque la communication bloquante est utilisée, la synchronisation est assurée par les composantes fournis de la plate-forme. Le concepteur peut cependant ajouter par lui-même des attentes explicites (`wait`) ou bien utiliser les objets de synchronisation de SystemC 2.0 comme les verrous ou les sémaphores. La figure 2.3 compare un cas de communication bloquante implicite avec un cas explicite.

	Communication bloquante	Communication non bloquante

Producteur	`while(true)`
	`{`
	`  m_port->write(...);`
	`}`
		`while(true)`
		`{`
		`  m_port->nb_write(...);`
		`  if (optionel)`
		`  {`
		`  wait( delai );`
		`  }`
		`}`

Consommateur	`while(true)`
	`{`
	`  int code = m_port->read(...);`
	`}`
		`while(true)`
		`{`
		`  while ( m_port->nb_read(...) == ERR )`
		`  {`
		`  wait( delai );`
		`  }`
		`}`

FIGURE 2.3 Comparaison : communication bloquante et non bloquante

L'accès aux périphériques est également possible avec l'interface du port de communication des modules. L'accès en lecture ou en écriture vers un périphérique est toujours bloquant, c'est-à-dire que le module qui effectue l'accès mémoire est bloqué jusqu'à ce que la requête soit traitée. Les instances de périphériques occupent tous une plage de l'espace mémoire de la plate-forme. Les adresses de début et de fin qui définissent ces plages mémoires sont des entiers non signés de 32 bits, doivent avoir un multiple entier de 32 bit de largeur et l'adresse de début de la plage doit être alignée sur 32 bits. Les plages d'adresses des périphériques ne peuvent pas se chevaucher, mais il peut y avoir des zones mémoires non définies, occupées par aucun périphérique. L'accès à ces zones cause évidemment des erreurs. La figure

2.4 résume ces règles d'adressage.

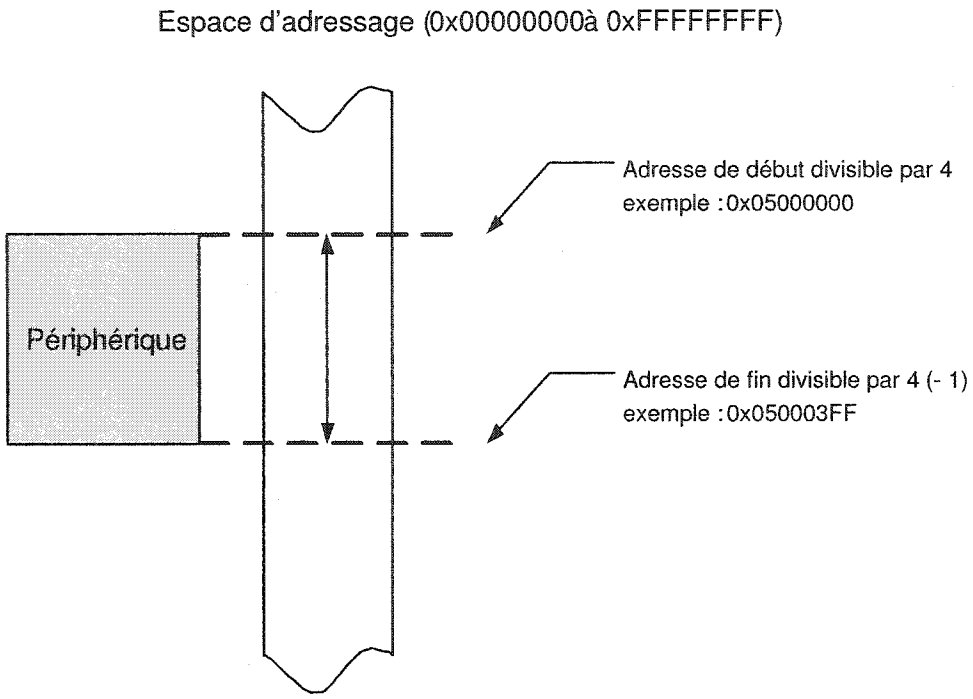

FIGURE 2.4 Restrictions sur les adresses des périphériques

Certaines adresses doivent être réservées pour les périphériques essentiels du système d'exploitation; ces adresses peuvent heureusement être changées d'une application à l'autre et doivent aussi être mises à jour lorsque la configuration de la plate-forme est changée (changement de processeur, ajout d'un périphérique, etc.). De plus, les processeurs RISC actuels utilisent souvent 32 bits pour l'adressage, et peuvent effectuer des accès mémoires limités à 32 bits également. Dans l'optique d'une compatibilité logicielle/matérielle des modules, une bonne idée est de restreindre la taille de la donnée lue ou écrite en mémoire à partir des modules à maximum 32 bits.

Le tableau 2.2 illustre un exemple de division de l'espace mémoire pour une instance de plate-forme.

TABLEAU 2.1 Exemple de division de la plage mémoire

Périphériques	Adresse de départ	Adresse de fin	Taille
RAM (code)	0x00000000	0x01FFFFFF	32 Mo
RAM (données)	0x02000000	0x03FFFFFF	32 Mo
RAM (vidéo)	0x04000000	0x047FFFFF	8 Mo
Adaptateur du processeur	0x04800000	0x04FFFFFF	8 Mo
Adaptateur des périphériques	0x05000000	0xFFFFFFFF	-
Minuterie	0x05000000	0x050003FF	1 Ko
Gestionnaire d'interruptions	0x05000400	0x050007FF	1 Ko
Périphérique d'arrêt	0xFFFFFFFC	0xFFFFFFFF	1 Ko

2.4.3 Niveaux d'abstraction

La plate-forme SPACE se situe au niveau d'abstraction TF. Notre méthodologie supporte également un niveau d'abstraction plus élevé : le niveau UTF. Le but est de minimiser les changements du code usager lors du passage du niveau UTF au niveau TF. Ceci accélère grandement le processus de raffinement. Le concepteur peut ainsi profiter du bienfait des deux niveaux d'abstraction sans avoir à adapter son code. L'avantage du niveau UTF est que la simulation est beaucoup plus rapide, car il n'y a pas tout l'arsenal pour faire exécuter le logiciel, ni la notion de temps de transfert des communications au niveau matériel. En fait, à ce niveau, il n'y a tout simplement pas de partitionnement logiciel/matériel. Tous les modules et les périphériques de l'application sont connectés ensemble sur un même canal : le `Glue Channel` (voir la figure 2.5). Comme à ce niveau il n'y a pas de distinction entre les instances de modules logiciels et matériels, la plate-forme ne requiert pas le système d'exploitation, ni tout les éléments et périphériques qui n'avaient aucune raison d'être mise à part d'assurer un bon fonctionnement de la partie logicielle : le gestionnaire d'interruption, la minuterie temps réel, le décodeur de l'ISS, l'adaptateur pour la communication ISS/modules, l'adaptateur pour la communication

ISS/périphériques et la mémoire vidéo conçue spécialement pour le logiciel. C'est d'ailleurs en partie grâce à cette simplicité que la simulation au niveau UTF est plus rapide.

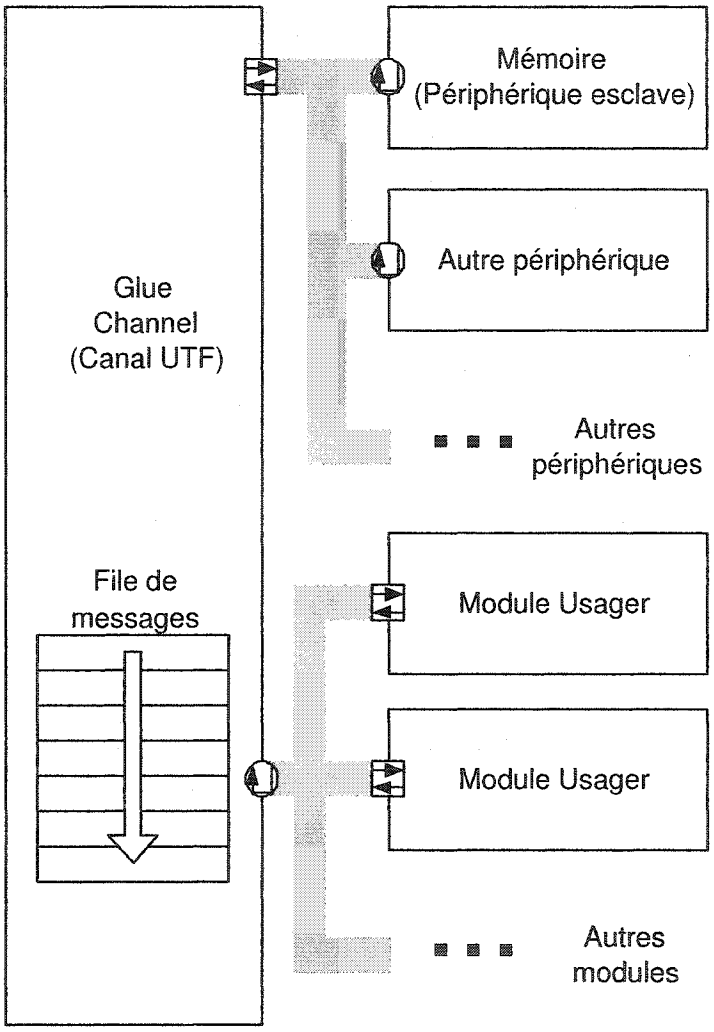

FIGURE 2.5 Schéma du Glue Channel

Un autre facteur qui accélère la simulation avec le `Glue Channel` est que les communications sont abstraites à un niveau où tout est intemporel. En effet, il n'y a aucun délai de transfert dans le `Glue Channel`, les lectures ou écritures sont effectués immédiatement et les processus ne sont bloqués que sur les événements d'envoi

ou de réception. La synchronisation demeure assurée. Évidemment, l'exécution ne respecte pas les cycles d'horloges, mais tel n'est pas l'objectif de ce niveau d'abstraction. Le niveau UTF est plutôt utile pour vérifier la fonctionnalité de l'application. C'est d'ailleurs l'approche que nous suggérons : une première conception de l'application peut être faite et vérifiée avec le `Glue Channel`. Lorsque la vérification fonctionnelle (validation) est terminée, le concepteur peut ensuite commencer à explorer les configurations logicielles/matérielles. Notre plate-forme SPACE procure l'avantage que les modules usagers n'ont pas besoin d'être modifiés pour passer du niveau UTF au niveau TF. Ceci demeure encore vrai pour revenir du niveau TF au niveau UTF : par exemple, il est possible que le concepteur veuille modifier son application en cours de route. À ce moment, il peut revenir au niveau UTF, faire ses modifications, vérifier que l'application fonctionne toujours et finalement refaire un partitionnement et continuer son exploration architecturale au niveau TF.

2.5 Structure de la plate-forme

Dans cette section nous allons décrire la structure de la plate-forme, en commençant par décrire les composants réutilisables préconçus et accessibles au concepteur.

2.5.1 Composants matériels fournis

Nous fournissons à l'utilisateur un choix de périphériques qu'il peut connecter pour construire son architecture. La figure 2.9 (placée à la fin du chapitre, à la page 49) présente l'ensemble des composants que nous allons décrire ici.

Processeur

Actuellement, notre choix s'est porté pour un ISS en langage C du ARM7.

Cet ISS est une version modifiée par GNU du Armulator de ARM [1]. Le fait d'utiliser un ISS de GNU a deux avantages : le premier est que la licence est gratuite et le code est ouvert. Le second est que GNU fourni un certain nombre d'ISS qui sont facilement interchangeables les uns les autres. Nous avons encapsulé le coeur de l'ISS dans un module SystemC (`SC_MODULE`). Cet ISS exécutera des instructions assembleur en un nombre précis de cycles d'horloge, de façon à respecter la spécification du processeur. Pour l'instant le signal d'horloge du processeur est le même que celui de la plate-forme.

Décodeur du processeur

Le processeur travaille en mode aligné et peut adresser un octet (8 bits), un mot (16 bits) ou un double mot (32 bits). Comme nous l'avons déjà mentionné, l'espace d'adressage est divisé entre les périphériques, qui occupent un intervalle d'adresse et les modules, qui ont un numéro d'identification avec une adresse équivalente. Le décodeur sert de multiplexeur et dirige les appels de fonction des interfaces SystemC vers le bon périphérique.

Adaptateur de modules du processeur

Pour faire communiquer notre ISS en C avec les modules matériels en SystemC, il faut un module effectuant la conversion entre les deux protocoles de communication. C'est-à-dire qu'il faudra passer d'une communication par adresses physiques et bits à une communication par messages et ports. L'adaptateur du processeur jouera ce rôle. Si les modules logiciels veulent envoyer des messages à des modules matériels, le décodeur active l'adaptateur du processeur. Ce périphérique occupe tout l'espace d'adressage des modules, et donc toutes les lectures ou écritures vers les modules passent par l'adaptateur. Comme le processeur ne peut pas envoyer ou recevoir plus de 32 bits à la fois, l'adaptateur sérialise (ou parallélise) les mots (ou les messages) et utilise des queues d'envoi ou de réception pour faire la conversion. De plus, dans le cas où le processeur et le simulateur SystemC de la plate-forme n'utilisent

pas le même ordre des octets dans les demi mots et mots (incompatibilité big-endian/little-endian), l'adaptateur du processeur pourra faire les conversion nécessaires (dans une version future, cela n'est pas implémenté pour l'instant).

Adaptateur de périphériques du processeur

Tout comme pour le cas des modules, les périphériques eux aussi sont adressés par un adaptateur qui se branche sur le décodeur du processeur. Lorsque les modules logiciels adressent les périphériques tels que la minuterie ou le gestionnaire d'interruptions, le décodeur active l'adaptateur, car il possède une plage d'adresse qui englobe toutes les adresses des périphériques du canal.

Adaptateurs des modules

Tous les modules de l'usager possèdent un port de communication pour envoyer et recevoir des messages. Cependant, la gestion des files de messages et de la mémoire pour les transferts bloquants et non bloquants est transparente pour le concepteur. Ce n'est pas à lui de gérer le tout, c'est la tâche des adaptateurs de modules. Pour chaque instance de module matériel, un adaptateur de module est nécessaire.

Mémoire de code

Elle contient le code binaire, qui regroupe le code du système d'exploitation, de l'interface SystemC et des modules et tâches logiciels. La mémoire de code est chargée à partir d'un fichier lors de l'initialisation de la simulation. La mémoire de code n'est pas accessible via le canal de la plate-forme. Seul le processeur peut avoir accès à cette mémoire via le décodeur.

Mémoire de données

Cette mémoire est utilisée pour stocker les informations dynamiques du logiciel. La mémoire de données possède à la fois l'interface du décodeur et des périphériques, du côté du canal. Ainsi, le processeur a accès à la mémoire en

passant par le décodeur, mais les modules matériels branchés sur le canal ont eux aussi accès aux données. Il est donc possible d'utiliser la mémoire à double port comme espace de rangement partagé. La cohérence des accès concurrentiels et les problèmes de corruption des données demeurent par contre une responsabilité du concepteur.

Mémoire vidéo

Nous avons conçu une mémoire vidéo qui permet au code logiciel d'envoyer des informations de déverminage à l'écran de l'ordinateur hôte où se déroule la simulation SystemC. Cette mémoire peut afficher des chaînes de caractères ou bien convertir des mots de 32 bits en chaînes numériques hexadécimales. Cette mémoire se branche sur le décodeur du processeur et aussi sur le canal, si le concepteur le désire. Cela permet entre autre aux modules d'utiliser un périphérique dédié pour l'affichage et le déboguage.

Gestionnaire d'interruptions

On insère ce périphérique entre les éléments qui émettent des interruptions et le processeur. En plus d'agir comme concentrateur (car il n'y a habituellement pas beaucoup de broches dédiées aux interruptions), cela centralise le traitement logiciel requis dans un seul périphérique.

Minuterie

Elle fournit les interruptions servant de base de temps au système d'exploitation temps réel qui s'exécute sur l'ISS. Les tics d'ordonnancement sont indispensables pour avoir un noyau préemptif tel celui de MicroC/OS-II. L'horloge source de la minuterie est l'horloge globale de la plate-forme.

Périphérique d'arrêt

Ce périphérique spécial permet de terminer la simulation. Un simple accès mémoire dirigé à l'adresse de ce périphérique engendre l'exécution de la fonction `sc_stop()` de SystemC.

Canal de la plate-forme

Il permet de raccorder tous les modules matériels et les périphériques de la plate-forme et assure les communications. Il est possible d'instancier plusieurs modèles (versions) du canal, pour émuler à haut niveau diverses structures ou protocoles. Entre autre, il est possible d'instancier un bus avec un arbitre.

2.5.2 Composants logiciels fournis

En plus des différents éléments matériels disponibles, nous offrons un support logiciel, composé de deux parties : le système d'exploitation MicroC/OS-II porté pour le ARM7 et l'interface SystemC qui émule le simulateur de SystemC dans lequel est implanté le gestionnaire de communication qui procure l'interface de communication SPACE aux modules logiciels. La figure 2.6 présente bien l'agencement des différents composants logiciels fournis. Voici donc une brève description des rôles de l'API, du gestionnaire de communication et du RTOS :

Interface SystemC

L'interface SystemC sert d'intermédiaire entre deux programmes, soit les modules usager de la plate-forme décrits en SystemC et le système d'exploitation. Cette interface a donc comme rôle principal d'interpréter le code d'une application et de convertir les appels SystemC en appels de fonctions systèmes propres au système d'exploitation. Par exemple, la création d'un processus SystemC se traduira par la création d'une tâche dans le système d'exploitation.

Gestionnaire de communication

Cette partie émule le comportement du canal matériel, pour les modules logiciels. C'est en fait le gestionnaire de communication qui implante la communication entre les tâches logicielles et entre les tâches logicielles et matérielles.

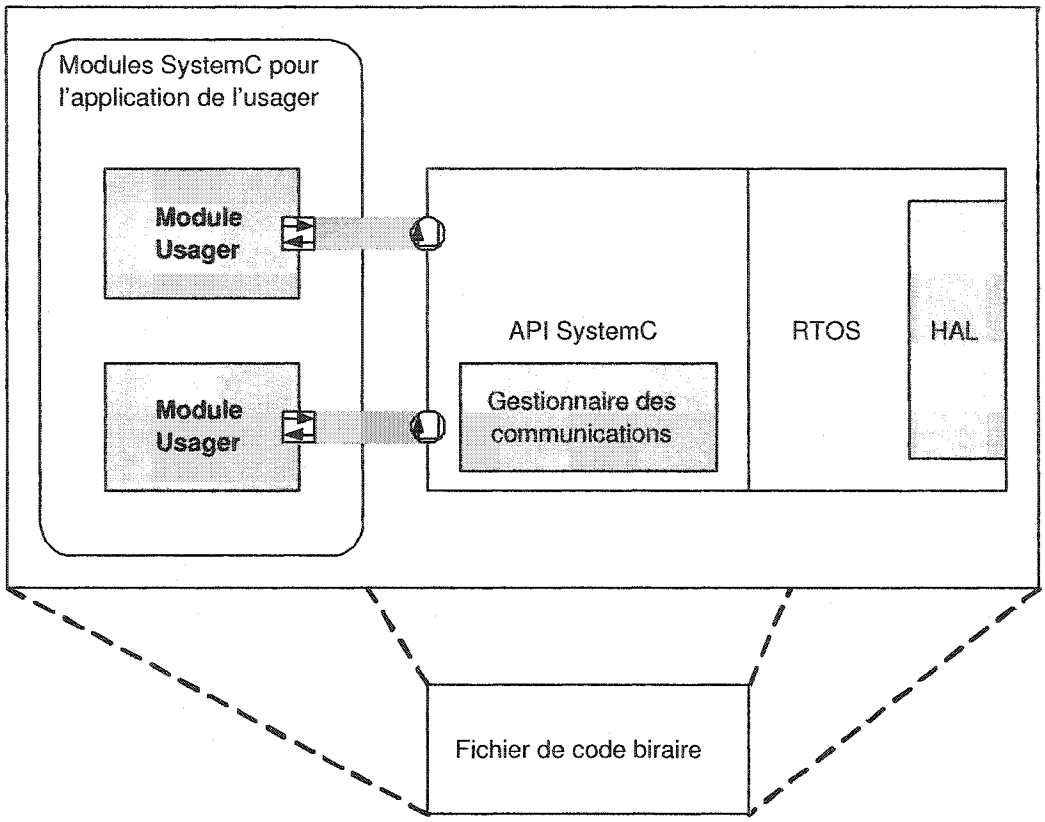

FIGURE 2.6 Schéma bloc de la partie logicielle

La synchronisation et le stockage des messages sont également assurés par le gestionnaire de communication, pour que l'environnement des modules logiciels soit le même que pour le cas des modules matériels.

Système d'exploitation

Il permet d'ordonnancer les différentes tâches des modules logiciels sous la forme de processus roulant sur l'ISS. De plus, nous avons également écrit des pilotes pour gérer et programmer les différents périphériques utilisés par le logiciel. Finalement, le port du système d'exploitation permet de traiter les interruptions.

2.5.3 Instances de plates-formes

Le concepteur peut instancier une plate-forme en connectant des instances de périphériques, des signaux de contrôle ou temporels, un processeur, mais aussi des instances de ses propres modules qu'il a conçu. Au niveau TF, les modules doivent être matériels ou logiciels. Pour chaque instance de plate-forme, il faut définir la configuration. Cette configuration comprend le partitionnement logiciel/matériel des modules, les ID des modules, les adresses des périphériques ainsi que certaines valeurs de configuration à fixer.

Au niveau UTF, l'application doit être partitionnée en un ou plusieurs éléments `SC_MODULE` et chacun des modules doit hériter d'une classe de base, définie pour le niveau UTF. Chaque module peut contenir un ou plusieurs `SC_THREAD`, cependant, dans le dernier cas chacun de ces processus doit communiquer avec d'autres modules différents (ou en d'autres mots, deux processus différents d'un même module ne peuvent pas communiquer avec un autre même module). Cette restriction vient du fait que les interfaces permettent de faire communiquer les modules entre eux et non pas les processus individuellement.

Au niveau TF, le concepteur peut réutiliser son code du niveau UTF sauf pour les exceptions qui suivent : au lieu d'utiliser des `SC_THREAD`, le concepteur doit utiliser des `SC_CTHREAD` et au lieu d'hériter de la classe de base `space_base_module` définie pour le niveau UTF, les modules de l'usager doivent hériter de la classe de base `space_base_module` définie pour le niveau TF. Comme le type de processus diffère au niveau UTF et au niveau TF, le concepteur doit utiliser la méthode `isHighLevel()` pour que son code ne soit pas modifié lors du passage UTF vers TF et vice-versa. Voici de quelle façon il doit procéder :

```
if ( isHighLevel() )
```

```
    wait( {nombre}, SC_NS ); // valide pour les SC_THREAD
else
    wait( {nombre} ); // valide pour les SC_CTHREAD
```

Ainsi, selon le niveau d'abstraction, l'une ou l'autre des fonctions `wait()` est appellée. Cela permet de réutiliser le même code pour les deux niveaux d'abstraction tout en respectant la syntaxe permise dans les processus de type `SC_THREAD` et `SC_CTHREAD`. Malheureusement, la condition est évaluée à l'exécution et non au moment de la compilation.

Nous allons donner ici un exemple d'instance de plate-forme au niveau UTF pour une application simple qui ne contient que deux modules. Ensuite nous verrons comment créer une plate-forme matérielle au niveau TF pour la même application. Cela va permettre d'expliquer par un exemple l'essentiel des règles architecturales pour les deux niveaux. Le détail sera donné à l'annexe IV.

Le bout de code ci-après présente les connexions de deux modules de l'usager, soit "producer" et "filter", branchés sur le canal `Glue Channel`. L'utilisateur doit d'abord définir l'unité de temps par défaut pour la simulation (1), ainsi que la résolution temporelle du simulateur. Ensuite, on instancie le canal et les modules de l'usager (2). Puis, on instancie un périphérque (3) : la mémoire de données. Il suffit ensuite de connecter les modules de l'usager au canal et de connecter le canal à la mémoire (4) pour pouvoir finalement démarrer la simulation (5).

```
// (1) Configuration temporelle
sc_set_time_resolution(1, SC_NS);
sc_set_default_time_unit(1, SC_NS);
// (2) Modules
glue_channel glue_channel1("glue_channel1");
```

```
producer producer1("producer1", ID_PRODUCER);
filter filter1("filter1", ID_FILTER);
// (3) Périphériques
ram_data ram_data1("ram_data1", RAM_START_ADDR, RAM_END_ADDR);
// (4) Connexions
producer1.m_port(glue_channel1);
filter1.m_port(glue_channel1);
glue_channel1.m_DevicePorts(ram_data1);
// (5) Début de la simulation
sc_start(-1);
```

La figure 2.7 présente une version graphique des connexions.

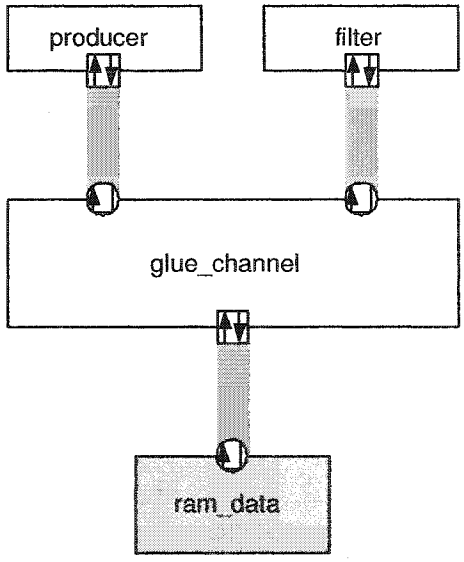

FIGURE 2.7 Connexions UTF

Pour la même application, l'instance de la plate-forme au niveau TF pourrait ressembler à ce qui suit (plusieurs détails sur l'implémentation de la partie matérielle seront vus au chapitre 3). La configuration temporelle (1) est la même qu'au ni-

veau UTF. Il faut ici ajouter le signal d'horloge ainsi qu'un signal pour une ligne d'interruption (2). Ce signal est requis, car nous avons ajouté à la plate-forme (4) l'adaptateur de modules du processeur (`iss_adapter1`). Toujours en (4), il faut aussi ajouter un composant bidon afin de pouvoir connecter le signal d'interruption inutilisé (voir la figure 2.8). Le canal `Glue Channel` a été remplacé par un canal TF de type bus. Nous voyons que les connexions requises (5) sont plus nombreuses, notamment parce qu'il faut connecter les modules de l'usager, les adaptateurs de module, le canal et l'adaptateur du processeur au signal d'horloge.

```
// (1) Configuration temporelle
sc_set_time_resolution(1, SC_NS);
sc_set_default_time_unit(1, SC_NS);
// (2) Signaux
sc_clock clk("clk", (1, SC_NS), 0.5);
sc_signal< bool > n_irq_iss_adapter;
// (3) Modules
producer producer1("producer1", ID_PRODUCER);
filter filter1("filter1", ID_FILTER);
// (4) Périphériques
module_adapter module_adapter1("module_adapter1", ID_PRODUCER);
module_adapter module_adapter2("module_adapter2", ID_FILTER);
iss_adapter iss_adapter1("iss_adapter1",
    ISS_ADAPTER_START_ADDR,
    ISS_ADAPTER_END_ADDR);
ram_data ram_data1("ram_data1", RAM_START_ADDR, RAM_END_ADDR);
space_channel_bus space_channel1("space_channel1");
dummy dummy1("dummy1");
// (5) Connexions
```

```
producer1.m_port(module_adapter1);
producer1.clock(clk);
filter1.m_port(module_adapter2);
filter1.clock(clk);
module_adapter1.m_PortToChannel(space_channel1);
module_adapter1.m_Clock(clk);
module_adapter2.m_PortToChannel(space_channel1);
module_adapter2.m_Clock(clk);
iss_adapter1.m_PortToChannel(space_channel1);
iss_adapter1.clk(clk);
iss_adapter1.n_IRQ(n_irq_iss_adapter);
space_channel1.m_AdapterPorts(module_adapter1);
space_channel1.m_AdapterPorts(module_adapter2);
space_channel1.m_DevicePorts(ram_data1);
space_channel1.m_SoftwarePort(iss_adapter1);
space_channel1.m_ClockPort(clk);
dummy1.m_DummyBoolPort(n_irq_iss_adapter);
// (6) Début de la simulation
sc_start(-1);
```

Comme mentionné précédemment, nous n'avons présenté ici qu'un sommaire des règles principales qui contraignent l'utilisateur, l'annexe III contient plus d'informations sur les règles d'implémentation.

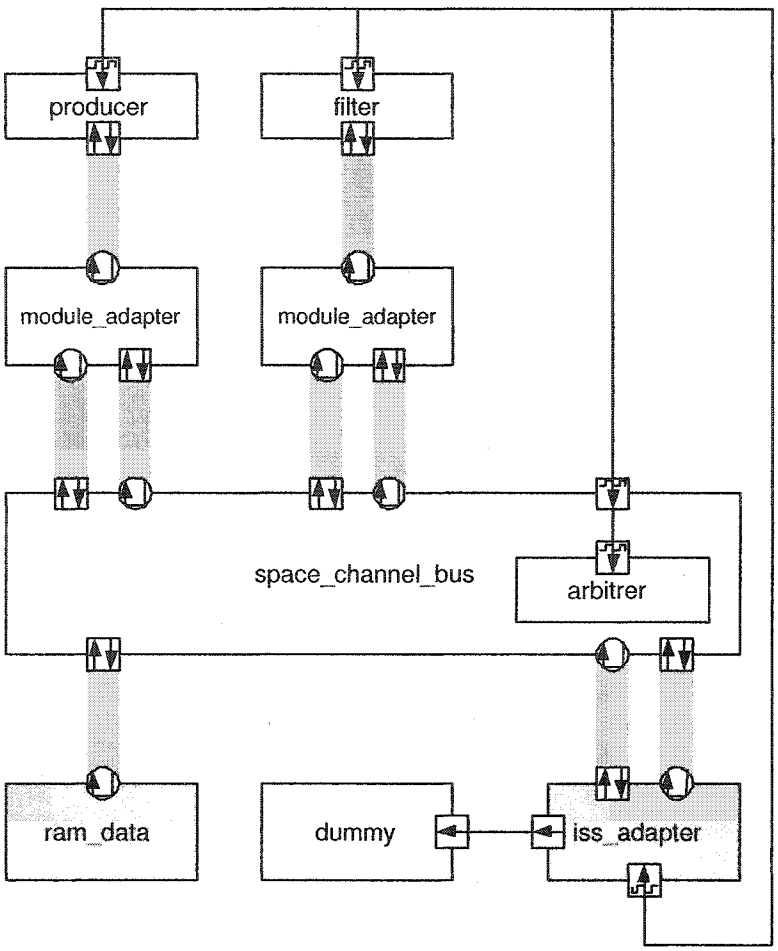

FIGURE 2.8 Connexions TF

2.6 Flot de conception (mode d'emploi pour un utilisateur)

La méthodologie SPACE est une approche descendante, c'est-à-dire que la conception d'une application débute avec une spécification, une implantation de haut niveau et des raffinements itératifs. Plus spécifiquement, un flot de conception avec approche descendante, dans la philosophie du codesign, ressemble à ceci :

1. Spécification fonctionnelle et non fonctionnelle du système, exploitation d'un

langage permettant d'exprimer le parallélisme intrinsèque ;

2. Choix de l'architecture cible ;
3. Affectation des tâches aux ressources de l'architecture choisie ;
4. Synthèse des communications ;
5. Synthèse logicielle ;
6. Synthèse matérielle.

Notre projet s'intéresse plus particulièrement aux étapes 1 à 4. L'étape 2 est relativement imposée par notre plate-forme et l'étape 3 est laissée aux mains de l'utilisateur.

Une fois les spécifications fonctionnelles écrites, le concepteur peut diviser son application en modules et en processus et implémenter une première version à haut niveau en SystemC. Le concepteur doit suivre la méthodologie SPACE dès le départ, en suivant les règles de conception précédemment décrites. Une fois la première version de l'application écrite, le concepteur effectue le branchement de ses modules avec le `Glue Channel`, le canal UTF de SPACE. Il doit également y connecter les périphériques requis et diviser l'espace d'adressage entre les périphériques. Les simulations UTF permettent au concepteur de vérifier la fonctionnalité de son application, de corriger ses erreurs de synchronisation, de conception et de syntaxe.

Une fois que le concepteur est en mesure de simuler correctement son application avec le `Glue Channel`, il doit procéder à un premier choix de partitionnement. Cette étape consiste simplement à choisir, au meilleur de sa connaissance les modules qui seront implémentés en logiciel et les autres qui seront en matériel. Ce choix n'est pas définitif ; au contraire, il pourra changer autant de fois que le concepteur le désire. Une fois le premier partitionnement établi, le concepteur doit créer une instance de plate-forme. Pour ce faire, il doit instancier ses modules matériels ainsi que tous

les autres périphériques nécessaires pour supporter le matériel et le logiciel. Il n'a qu'à instancier les éléments en SystemC, effectuer les branchements et compiler le tout. Concernant les modules logiciels, il doit les connecter à l'émulateur de SystemC (API SystemC). Le concepteur peut jouer sur les paramètres logiciels, par exemple l'adresse du gestionnaire d'interruptions ou bien celle de la minuterie. Ensuite il doit compiler le système d'exploitation ainsi que la couche API SystemC avec sa configuration. Une fois tout le logiciel compilé, un fichier binaire exécutable est obtenu. Ce fichier est utilisé en paramètre de la simulation de la plate-forme matérielle. Il est alors possible d'exécuter l'application qui est désormais matérielle et logicielle.

Lorsque la simulation est lancée, cette fois au niveau TF, les parties logicielles et matérielles s'initialisent et exécutent le code de l'application. SystemC exécute le code matériel et donne la main au processus de l'ISS pour lui permettre d'exécuter le code binaire passé en paramètre. Cette fois les notions de temps sont prisent en considération lors de la simulation, ce qui permet de fournir des résultats intéressants au concepteur. Par contre, le temps d'exécution est affecté. La communication qui était vérifiée au niveau UTF fonctionne toujours, car notre plate-forme s'occupe automatiquement de la synthèse (limitée) des communications. La synthèse des communications proprement dite consiste à implémenter les canaux de communications permettant les échanges de données entre les processus [22, 11]. Notre méthodologie est limitée au niveau TF et la synthèse des communications s'arrête à ce niveau.

Pour le moment, la métrique choisie pour évaluer le choix de partitionnement est le nombre de cycles d'horloge pour exécuter l'application ou une partie de l'application. Bien souvent, une application embarquée n'a pas de fin, elle s'exécute de façon perpétuelle. Dans ce cas, le concepteur doit utiliser une condition pour mettre fin à la simulation. Il doit choisir cette condition de façon à ce qu'elle soit valide peu

importe le partitionnement logiciel/matériel choisi et peu importe le niveau de la simulation (i.e. UTF ou TF). Le concepteur peut par conséquent définir un endroit où l'application s'interrompt et mesurer le nombre de cycles d'horloges utilisés jusqu'à ce moment précis. Le nombre de cycles peut être comparé à des contraintes temps réel définies dans la spécification. Dans le cas où le partitionnement choisi ne respecte pas les contraintes de temps, il est possible de le changer et de refaire des simulations avec la nouvelle configuration logicielle/matérielle. Pour ce faire, il suffit de reprendre les étapes de partitionnement et de refaire des simulations. Ce processus n'est pas ardu, mais peut prendre un peu de temps et peut aussi être une source d'erreur. Si les contraintes de temps sont respectées, le concepteur peut tenter de réduire le nombre de modules matériels en les transférant du côté logiciel.

En ayant en tête une future implémentation physique de l'application, nous pouvons comprendre que l'augmentation du nombre de modules logiciels n'accroît pas autant la surface que l'augmentation du nombre de modules matériels de l'application. En effet, peu importe le nombre de tâches logicielles que pourra comporter l'application, il n'y aura jamais plus d'un processeur pour exécuter tout ce code. L'augmentation du nombre de modules logiciels n'augmente principalement que la quantité de mémoire requise pour l'exécution. Neanmoins, la puissance dissipée du processeur pourrait varier, selon le nombre de tâches logicielles et le type d'application. Il y a des études [17, 27] qui montrent que le type de programmation peut avoir un impact sur la puissance dissipée d'un processeur.

Par opposition, l'addition de modules matériels requiert l'ajout de connections matérielles vers le canal, l'augmentation de puissance électrique consommée, ajoute des tampons mémoires coûteux en surface en plus de la surface pour placer ces modules sur le système sur puce. Ainsi, nous pouvons définir le partitionnement logiciel/matériel optimal comme étant la configuration pour laquelle les modules sont le plus possible dans la partie logicielle, tout en laissant les modules qui assurent

la rencontre minimale des contraintes de temps en matériel.

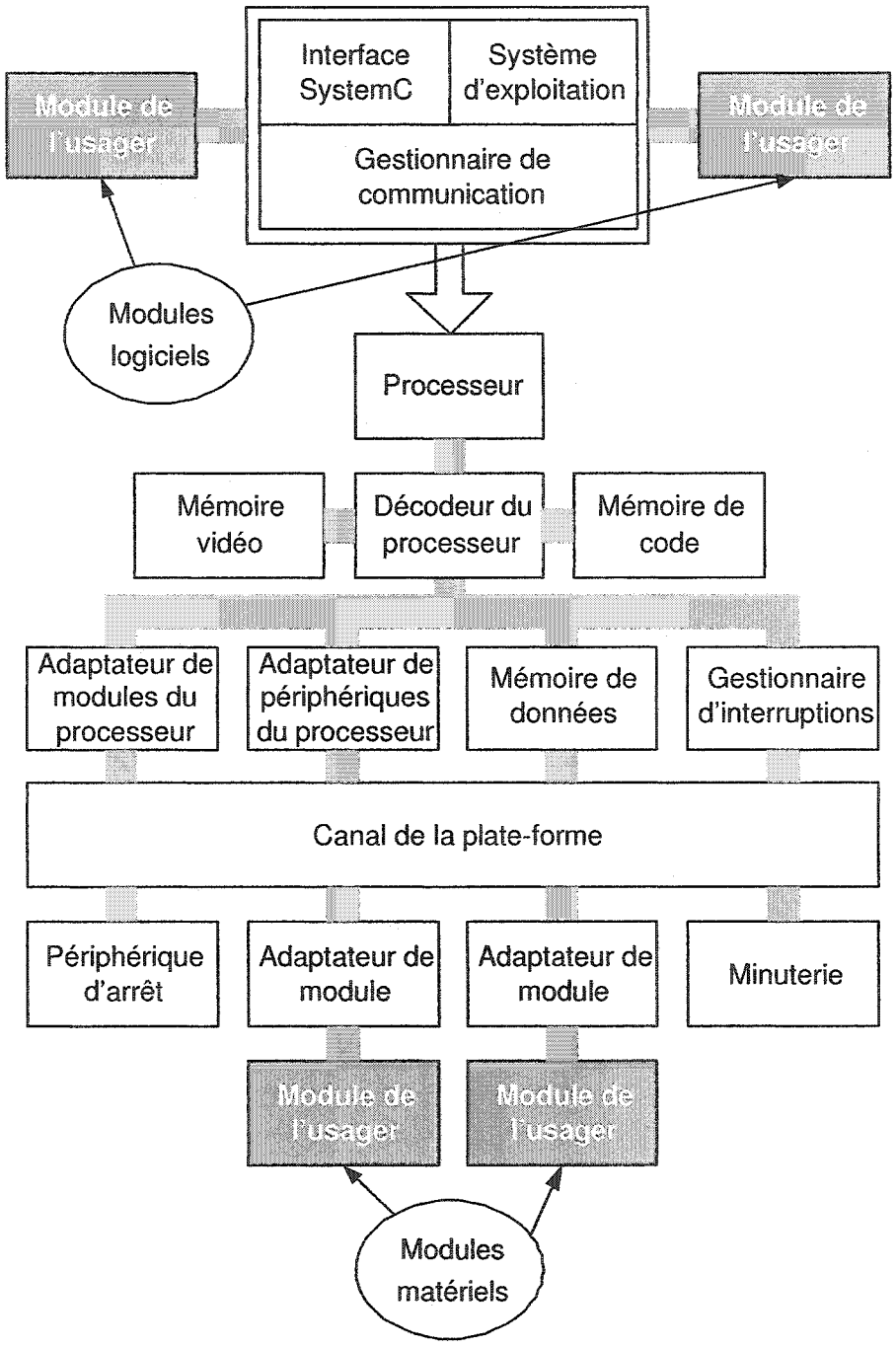

FIGURE 2.9 Aperçu des composants de la plate-forme au niveau TF

CHAPITRE 3

IMPLÉMENTATION DES COMMUNICATIONS

3.1 Fonctionnement général

Il est maintenant temps de décrire le fonctionnement des communications sur la plate-forme. Dans un premier temps, nous allons aborder le sujet d'un oeil d'utilisateur et ensuite nous entrerons dans les détails d'implémentation.

3.1.1 D'un point de vue de l'utilisateur

Les cycles d'horloges ne sont pas apparents pour l'utilisateur, car ils sont encapsulés dans les appels de fonction pour la communication. De plus, l'ordre d'exécution des processus n'est pas contrôlé par le concepteur, c'est donc la responsabilité du canal de communication d'assurer la synchronisation. Rappelons-nous que le simulateur de SystemC exécute de façon concurrente les processus, mais émule un comportement parallèle, de façon identique à n'importe quel simulateur matériel (exemple ModelSim [28] pour le VHDL). Voir l'annexe I et [30, 7] pour les détails sur le comportement du simulateur de SystemC.

3.1.2 Communication module à module (UTF)

Prenons l'exemple d'une application très simple : un module producteur et un module consommateur. Chacun de ces modules ne contient qu'un seul processus (`SC_THREAD`). Le module producteur effectue une écriture, c'est-à-dire qu'il envoie

une donnée et ensuite se met en attente pour la prochaine exécution. Le module consommateur n'est guère plus complexe ; il effectue une lecture pour recevoir la donnée et se met en attente pour la prochaine exécution. Si ces modules sont conçus en SystemC et que les modules sont connectés ensemble avec une file de message bloquante (FIFO) d'une profondeur de 1 (c'est-à-dire qu'un seul message peut voyager du producteur vers le consommateur à la fois), nous obtenons l'application producteur/consommateur la plus simple. Si les communications sont bloquantes, le consommateur sera interrompu si la donnée n'est pas produite lors d'une lecture (FIFO vide) et le producteur sera également interrompu si la donnée n'a pas été consommée (FIFO plein). Nous voyons que le simulateur de SystemC assure la synchronisation entre les processus.

Maintenant, si nous prenons nos deux modules et que nous les branchons sur le canal UTF avec d'autres modules, qu'arrivera-t-il ?

Puisque les modules ont été conçus de façon à être synchronisés par les communications, nous proposons comme solution que l'exécution des tâches se fasse sans préemption entre deux appels de fonction de communication :

```
read(...)
... Exécution sans préemption
write(...)
```

Notre motivation pour ce type de modèle est la suivante. L'exécution du code des modules logiciels va consommer des cycles d'horloge, parce que le processeur prendra un nombre précis de cycles pour chaque instruction assembleur. Les modules matériels sont considérés comme de puissantes unités de calcul et leur exécution consommera des cycles lorsqu'ils communiqueront ou lorsque le concepteur emploiera explicitement la fonction `wait()` de SystemC. Dans tous les autres cas, le

concepteur n'aura pas à spécifier de latence d'exécution entre les appels de communication.

Une autre chose à considérer est l'ordre d'exécution. Il faut tenir compte du fait que n'importe quel module peut s'exécuter en premier et il faut s'assurer que la synchronisation est respectée. Pour ce faire, nous avons défini une structure interne particulière (une queue de messages). Dans ce qui suit, nous allons expliquer le tout avec un exemple.

La situation est celle de plusieurs producteurs et de plusieurs consommateurs. Un producteur envoie une donnée à un consommateur précis grâce au numéro d'identification unique de ce dernier. Ce message est écrit dans la file du canal de communication. Lorsque le consommateur demande une lecture avec la fonction `read()`, il identifie lui aussi le numéro du producteur visé ainsi que son numéro d'identification unique. Si la file de message du canal possède déjà un message qui est destiné à ce consommateur particulier et qui est en provenance du bon producteur, alors le message est directement transmis au consommateur qui peut poursuivre sont exécution. Cependant si aucun message n'est disponible, alors la requête de lecture est placée dans la file d'attente et le consommateur bloque en attendant une réponse. La même chose se produit si un producteur écrit un message pour lequel aucun consommateur n'est en attente, c'est-à-dire que le message provenant du producteur est placé dans la file d'attente et que le producteur se bloque en attendant que son message soit consommé. Il s'agit donc d'une communication sans horloge synchronisée sur les données.

3.1.3 Communication module à périphérique (UTF)

La communication entre un module et un périphérique est plus simple puisqu'il s'agit d'une relation de maître à esclave où le périphérique ne fait que répondre à des requêtes. Il n'y a donc pas de problèmes liés à la concurrence.

3.1.4 Communication module à module (TF)

Nous savons que les modules peuvent passer d'une nature logicielle à matérielle très facilement, avec très peu d'effort de la part du concepteur. La communication doit cependant être garantie et nous allons voir comment tout cela peut être possible.

Les modules matériels ne sont pas connectés sur le canal de la plate-forme directement, il y a toujours un adaptateur qui doit se placer entre les deux. Ce composant contient une liste de messages qui joue le rôle de cache de messages ou boîte de réception de messages. En comparaison avec le canal UTF, le canal TF ne contient pas de listes. Le canal effectue le routage des messages et c'est dans les adaptateurs que s'entreposent les messages.

Les files de messages n'ont pas été placées à même les modules de l'usager parce que les listes sont traitées différemment dans le cas du logiciel et dans le cas du matériel. Il fallait donc les extraire pour ne pas laisser le traitement des messages à l'usager et rendre le tout transparent. Bien que les files de messages auraient pu être implantées directement dans le canal, nous désirions avoir des mémoires séparées pour chaque module, de façon à pouvoir mieux estimer la quantité de mémoire requise pour chaque module et aussi pour avoir une approche modulaire, question d'extraire une fonctionnalité du canal et d'en simplifier sa conception. Contrairement au niveau UTF, au niveau TF il est intéressant de pouvoir changer le

modèle du canal en conservant les adaptateurs intacts. Pour développer un nouveau protocole de communication, le concepteur n'a pas à refaire les boîtes de réception de message (adaptateurs). Par contre, s'il le désire, il peut bien changer le modèle des adaptateurs sans toucher à la fonctionnalité du canal.

Lorsqu'un module de l'application effectue une lecture, l'adaptateur vérifie s'il y a un message dans la boîte. S'il n'y a pas de message, la fonction de lecture retourne un message d'erreur ou bloque la tâche et se met en attente d'un message, dépendamment si le concepteur a utilisé une fonction bloquante ou non. Si la boîte à messages n'est pas vide, le module fait la lecture du message directement dans la boite. Si le module effectue une opération d'écriture, c'est à ce moment qu'une transaction sur le canal est initiée. Le canal fait son travail et le message se retrouve dans la mémoire de messages du module récepteur. Nous voyons donc que seules les opérations d'écriture produisent des transactions sur le canal. Ceci nous permet d'attribuer au matériel un comportement similaire au logiciel. Chaque module matériel aura sa mémoire de messages et chaque tâche logicielle aura sa file de messages, gérée par le système d'exploitation. De plus, nous réduisons le trafic sur le canal en laissant les adaptateurs se charger de retenir les opérations de lecture.

3.1.5 Communication module à périphérique (TF)

L'accès aux périphériques avec le canal TF fonctionne de la même façon qu'avec le canal UTF. Cette fois par contre, les accès mémoire consomment des cycles d'horloge (temps de transfert dans le canal) et sont toujours bloquants, c'est-à-dire que le module qui fait un accès vers un périphérique doit attendre que ce dernier lui réponde avant de pouvoir continuer à s'exécuter.

3.2 Interfaces

La définition des interfaces dans SPACE nous permet de déterminer les rôles, les permissions et les compatibilités entre les différentes entités de la plate-forme. Chaque interface reflète les fonctionnalités disponibles d'un port ou d'un composant, i.e. que les modules usager peuvent faire des lectures, des écritures, des opérations bloquantes, non bloquantes et des accès mémoire, tandis que les périphériques ne font que répondre aux accès mémoire, les adaptateurs sont là pour retenir les lectures et laisser passer les écritures et finalement sur le canal il n'y a que les écritures qui voyagent. Avec ces quelques règles simples, on peut définir toutes les interfaces. Il y a aussi une interface pour le processeur (l'ISS), puisque le processeur fonctionne par adresses et données. La figure 3.1 présente l'ensemble des interfaces de SPACE.

Nous allons présenter les différentes interfaces en détails avec chacune leur appartenance pour les ports et pour l'implémentation (i.e. quel composant peut avoir un port de ce type d'interface et quel composant peut implémenter une interface de ce type), le jeu de méthodes qui compose chacune des interfaces ainsi qu'une brève description.

TABLEAU 3.1 Interface space_module_if

space_module_if	
Description	Interface disponible aux modules de l'usager
Port	Modules de l'usager
Implémentation	glue_channel, module_adapter
Méthodes	read(), write(), nb_read(), nb_write(), mem_read(), mem_write()

Dans les tableaux, nous pouvons comprendre que les composants listés dans la section "Port" peuvent être connectés à ceux qui sont listés dans la section

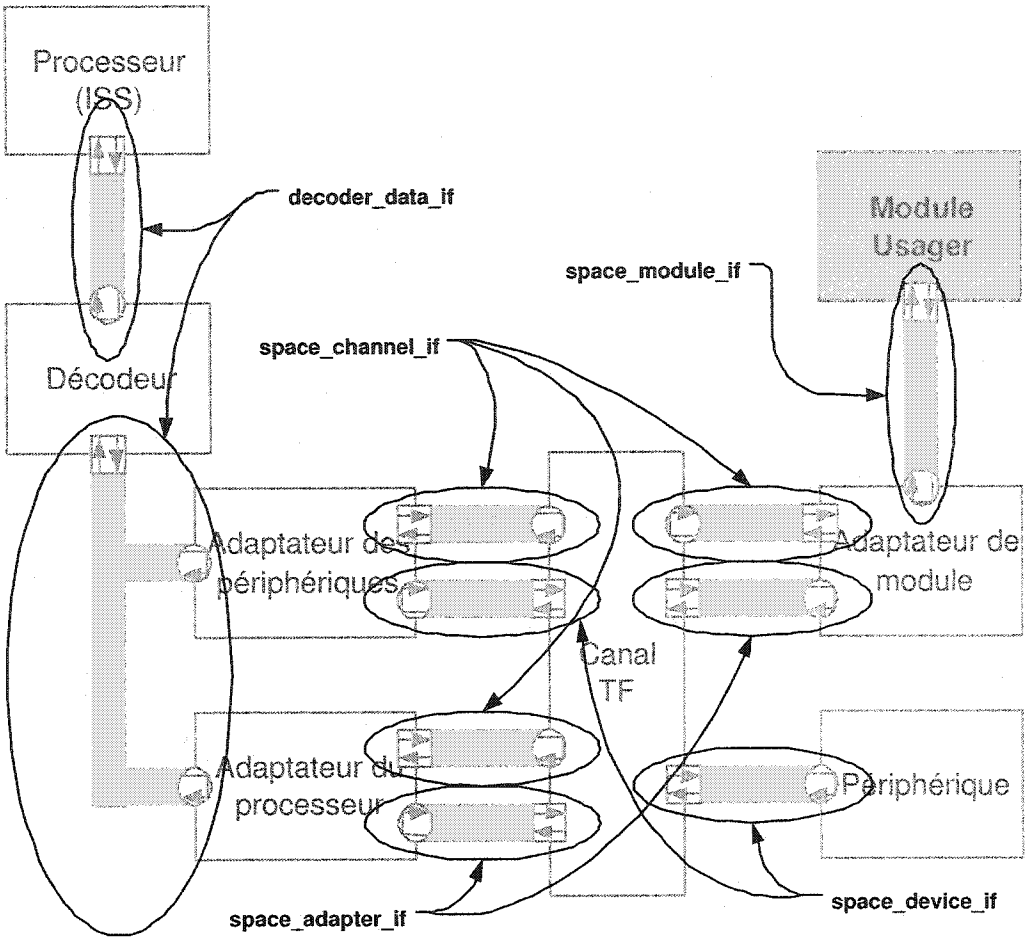

FIGURE 3.1 Interfaces SystemC pour la plate-forme

"Implémentation". En d'autres mots, il est possible de brancher un port P d'un module A à une interface I du module B si le port du module A utilise l'interface I et que le module B implémente l'interface I.

Nous pouvons comparer nos interfaces avec celles de SOCP de StepNP [40]. Les deux utilisent les constructions de SystemC 2.0 et permettent d'opérer au niveau fonctionnel/transactionnel. Avec SOCP, toutes les transactions sont interrompues ou divisées (splitted) par défaut. Avec SPACE, rien de tel n'a été prévu, c'est-à-dire que le canal peut implémenter cette fonctionnalité, mais l'usager n'a pas à

TABLEAU 3.2 Interface space_device_if

space_device_if	
Description	Interface que doivent implémenter les périphériques
Port	glue_channel, space_channel
Implémentation	space_base_device (tous les périphériques), device_adapter
Méthodes	mem_read_from_channel(), mem_write_from_channel(), start_address(), end_address()

TABLEAU 3.3 Interface space_adapter_if

space_adapter_if	
Description	Interface que doivent implémenter les adaptateurs qui se branchent sur le canal
Port	space_channel
Implémentation	module_adapter (adaptateurs de tous les modules), iss_adapter
Méthodes	write_from_channel(), nb_write_from_channel()

faire deux appels de fonction pour envoyer une requête et récupérer la réponse ; tout cela peut être caché dans les adaptateurs. Une autre fonctionnalité qui diffère est le support des maîtres et esclaves multi processus (multithreaded). Cela se reflète au niveau des interfaces de SOCP avec la présence d'identificateurs de processus (thread) en plus des identificateurs d'entités (ID numbers). Ce support multi processus permet à un module d'envoyer une requête à un autre module en s'adressant

TABLEAU 3.4 Interface space_channel_if

space_channel_if	
Description	Interface que fournit le canal ; tous les composants qui s'y branchent doivent utiliser un port de ce type
Port	device_adapter, iss_adapter, module_adapter
Implémentation	space_channel
Méthodes	write(), nb_write(), mem_read(), mem_write()

TABLEAU 3.5 Interface decoder_data_if

decoder_data_if	
Description	Interface qu'utilise le processeur (l'ISS) pour les accès mémoire ; tous les composants qui peuvent se brancher sur le décodeur doivent implémenter cette interface
Port	iss, decoder
Implémentation	decoder, device_adapter, iss_adapter, ram_data, ram_code, video_ram
Méthodes	readword(), writeword(), readhword(), writehword(), readbyte(), writebyte(), start_address(), end_address()

précisément à un seul processus membre de ce module destinataire. Bien que cette fonctionnalité soit intéressante, elle ajoute un degré de complexité non négligeable dans l'implémentation des communications. Pour cette raison, nous avons omis cette caractéristique de la définition de nos interfaces. Finalement, une dernière fonctionnalité disponible dans les interfaces de StepNP est manquante dans notre implémentation de SPACE : le mode rafale. Cela permet à un module d'envoyer plusieurs requêtes successives. Avec une implémentation de canal de type bus, le mode rafale consiste à verrouiller l'accès au bus par un seul maître. Après avoir fait ses requêtes en rafale, le verrou est relâché et l'arbitre du bus peut donner l'accès à un autre maître. Tel est le fonctionnement de Simple Bus [20].

3.3 Composants pour la communication

Cette section décrit un par un les composants essentiels pour la communication, en donnant leur spécification et en survolant rapidement leur fonctionnement interne. Le lecteur peut se référer à la figure 3.2, qui présente une instance assez générique de la plate-forme, contenant les éléments que nous allons décrire.

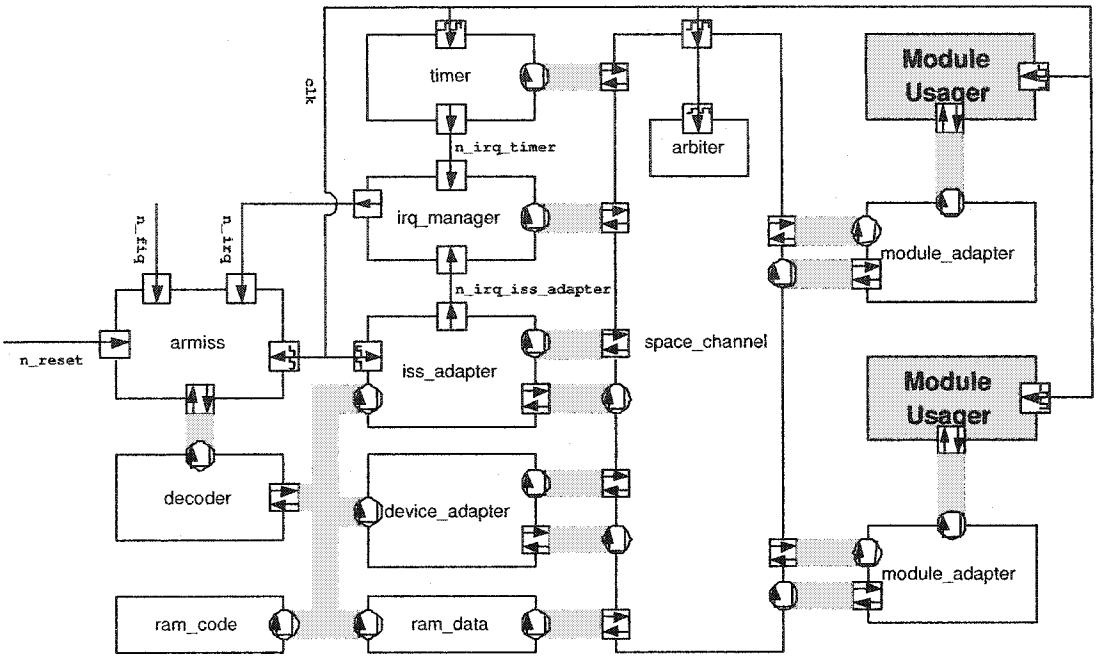

FIGURE 3.2 Architecture matérielle générale pour SPACE

3.3.1 Composant glue_channel

Le `glue_channel` est le canal de communication pour le niveau UTF. Il sert à faire la liaison entre les modules, en assurant une communication fonctionnelle (voir la figure 2.5 au chapitre 2). Pour l'utiliser, il faut simplement connecter les modules de l'usager et les périphériques voulus.

Comme nous l'avons mentionné déjà, les modules de l'usager sont les seuls qui peuvent initier des transactions. Quand une requête de lecture ou d'écriture est envoyée, le `glue_channel` utilise une liste de requêtes interne pour savoir s'il s'agit d'une nouvelle requête ou pour répondre à une requête qui était déjà en attente. La synchronisation est assurée par des événements de SystemC (`sc_event`).

3.3.2 Composant space_base_module

Aux niveaux UTF et TF, les modules usager doivent hériter de cette classe. Cela permet d'imposer des éléments structurels communs et obligatoires à tous les modules, dépendamment du niveau d'abstraction. Alors qu'au niveau UTF, il faut connecter le module sur le `glue_channel`, au niveau TF il faut le brancher à un adaptateur de module. Ces deux composants sont compatibles avec le port défini dans `space_base_module`. De plus, au niveau TF, il faut connecter le module sur l'horloge globale. Dans le cas où l'on désirerait avoir un comportement différent au niveau UTF et au niveau TF, on peut utiliser la méthode `isHighLevel()` qui retourne vrai (`true`) au niveau UTF et faux au niveau TF (`false`).

À l'interne, le port unique de communication pour les modules est défini dans la classe de base. Il y a véritablement deux versions de classe `space_base_module` : une pour le UTF et l'autre pour le TF. La différence entre les deux (mis à part le fait que la méthode `isHighLevel()` retourne une valeur différente) est qu'au niveau TF il y a en plus le port pour l'horloge. Ce port permet de créer des processus de type `SC_CTHREAD`.

3.3.3 Composant space_base_device

Tout comme pour les modules, les périphériques doivent eux aussi hériter d'une classe de base : `space_base_device`. Lors de l'instanciation d'un périphérique, il faut passer en paramètre du constructeur l'adresse de début et l'adresse de fin de la plage d'adresse voulue du périphérique. Le concepteur doit implémenter les méthodes de l'interface des périphériques, pour définir les comportements en cas de lecture ou d'écriture.

Il est possible d'ajouter aux périphériques un port d'horloge, si désiré, car la classe de base n'en contient pas. Dans le cas où on désirerait avoir un comportement différent au niveau UTF et au niveau TF, on peut utiliser la méthode `isHighLevel()` qui retourne vrai (`true`) au niveau UTF et faux au niveau TF (`false`), comme cela a été expliqué pour le cas des modules.

3.3.4 Composant module_adapter

Les figures 3.3 et 3.4 décrivent respectivement les vues internes des deux versions possibles du `module_adapter` : sans processus et avec processus, dont les différences sont expliquées un peu plus loin. Le concepteur peut choisir l'une ou l'autre des versions au moment de la compilation.

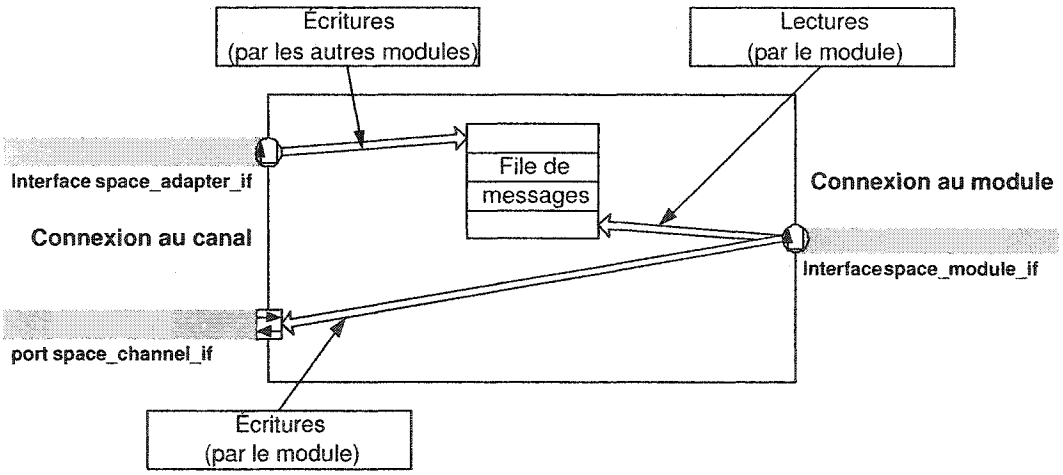

FIGURE 3.3 Fonctionnement du périphérique module_adapter

L'adaptateur de module sert à effectuer la synchronisation entre les différents modules matériels de l'usager. Sur ces mêmes figures, on observe bien qu'il n'y a que les écritures qui voyagent sur le canal ; les adaptateurs servent à stocker ces requêtes d'écriture.

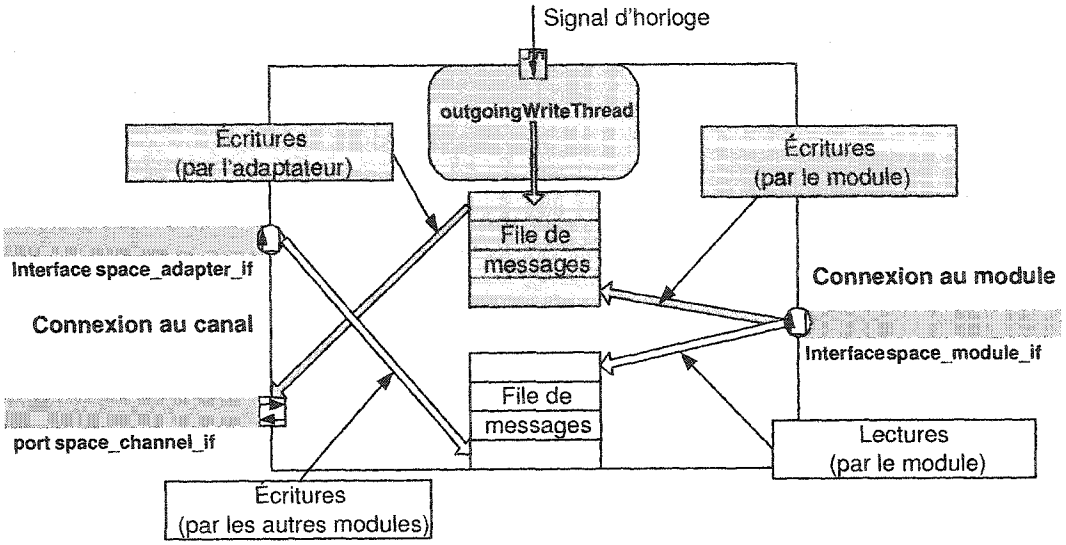

FIGURE 3.4 Fonctionnement du module_adapter avec processus

L'interface `space_adapter_if` permet de brancher le canal vers l'adaptateur, i.e. que les requêtes provenant du canal (des autres modules) "arrivent" par cette interface. L'interface `space_module_if` permet de connecter un module usager à l'adaptateur. L'adaptateur est muni d'un port de type `space_channel_if`, pour brancher l'adaptateur au canal, lui permettant d'initier des requêtes vers le canal, pour faire des écritures vers d'autres modules. Si on choisit la version avec processus (figure 3.4), un port d'horloge est aussi présent. Cela permet à l'adaptateur d'avoir un processus synchrone.

Les deux versions du `module_adapter` se comportent de la même façon pour les requêtes de lecture et pour les opérations d'entrées/sorties sur la mémoire, c'est-à-dire pour les méthodes `mem_read()` et `mem_write()`. Ces dernières sont toujours directes au canal et l'adaptateur ne fait que passer la requête du module vers le canal.

Lors d'une requête de lecture, l'adaptateur consulte sa liste interne de requêtes. Si

une requête d'écriture correspondante est déjà dans la liste en attente, l'adaptateur transmet le message dans la réponse à la lecture. Si la requête correspondante n'est pas encore arrivée, l'adaptateur retourne un code d'erreur si la lecture était non bloquante, sinon on attend le cycle suivant pour consulter la liste, dans le cas bloquant.

En comparant les schémas des figures 3.3 et 3.4, nous voyons que le fait d'ajouter le processus synchrone `outgoingWriteThread()` permet d'envoyer les requêtes d'écriture de façon indépendante des modules. En fait, comme le canal effectue des appels à la fonction de synchronisation `wait()`, le concepteur peut vouloir éviter que les modules soit bloqués pour quelques cycles d'horloge durant un appel d'écriture non bloquante. En utilisant la version avec processus, c'est l'adaptateur qui se charge d'initier la transaction en effectuant lui-même l'appel de méthode `nb_write()` ou `write()`, évitant à ce moment de bloquer le processus du module qui avait initialement fait une requête.

3.3.5 Composant space_channel

Le canal TF est construit sur une base qui facilite l'expansion et l'exploration architecturale. Une classe `space_channel` est donc fournie comme classe de base et plusieurs canaux avec des protocoles ou architectures différents peuvent être construits à partir de ce squelette. De cette façon, tous les canaux qui sont construits respectent les interfaces fournies et cela assure une compatibilité entre les différents canaux, les rendant interchangeables.

La classe de base comporte deux multi-ports, un pour y brancher les adaptateurs et un autre pour y brancher les périphériques. Des méthodes membres permettent de retrouver les éléments connectés à ces ports lors de l'initialisation en simulation,

pour les indexer dans une table ou cache interne. Des méthodes sont aussi fournies pour ensuite les retrouver à partir des ID (dans le cas des adaptateurs de module) ou des adresses (dans le cas des périphériques) :

```
void scanDevices(void);
unsigned long getDeviceNumber(const unsigned long Address);

void scanAdapters(void);
unsigned long getAdapterNumber(const unsigned long ID);
```

L'implémentation actuelle de SPACE fournie deux canaux : un réseau point à point complet (crossbar) et un bus. Leurs caractéristiques seront décrites aux deux prochaines sous-sections.

3.3.6 Composant space_channel_xbar

Il s'agit d'un canal qui modélise un réseau où tous les modules ont un accès point à point avec les autres modules. On peut y brancher un adaptateur spécial, le périphérique `iss_adapter`, qui à son tour permet de brancher un modèle de processeur au canal. Dans ce cas, il a été défini qu'on attribut le numéro unique (ID) de valeur zéro à l'`iss_adapter`. C'est pour cette raison que les modules de l'usager ne devraient pas utiliser ce numéro. Le réseau modélise une largeur de 32 bits et donc un délai (latence) de 1 cycle est attribué au transfert de chaque bloc de 32 bits qui est transféré, que ce soit avec une écriture bloquante, une écriture non bloquante, une lecture ou une écriture en mémoire.

3.3.7 Composant space_channel_bus

Contrairement au réseau point à point, le bus possède un port dédié pour le périphérique `iss_adapter`. Pour des raisons de compatibilité entre les deux types de canaux, l'usager ne devrait tout de même pas utiliser le numéro d'identification zéro pour ses modules, même s'il utilise le bus.

Une autre différence structurelle avec le bus est l'ajout d'un arbitre intégré. L'arbitre possède un processus synchrone (`SC_CTHREAD`), c'est pourquoi un port d'horloge a été ajouté au canal. La figure 3.2 illustre comment le tout est connecté. Le fait d'ajouter un arbitre complexifie le traitement des requêtes dans le canal. Voici plus de détails sur comment sont traités les requêtes d'écriture.

1. D'abord, il faut demander à l'arbitre l'accès au bus. Ceci s'effectue au moyen d'un appel de fonction qui se bloque si le canal est occupé. Cela a pour effet de mettre le processus qui fait la requête en attente. À chaque front montant du cycle d'horloge, la fonction vérifie si le processus a obtenu le droit de transfert.
2. L'arbitre possède un processus synchrone qui est actif sur le front descendant de l'horloge, un peu comme cela a été fait dans Simple Bus [20]. L'algorithme d'arbitrage dicte que chaque processus doit attendre au moins un cycle pour avoir le bus, pour simuler un délai de traitement. Lorsqu'un processus demande le droit d'accès à l'arbitre, il fourni son ID ainsi qu'un nombre correspondant à une priorité. L'arbitre classe alors les requêtes d'accès en ordre de priorité dans une queue. À chaque cycle d'horloge (au front descendant), tous les ID des processus qui ont fait la demande d'accès ont été classés dans la queue de priorités. Les requêtes ordonnées sont alors déplacées à la fin d'une liste simple des requêtes existantes. Lorsqu'une requête se termine et libère

le bus, l'arbitre en est informé et récupère la requête qui se trouve en début de liste et donne le bus au demandeur correspondant, qui pourra commencer le transfert au front montant suivant. De là vient la latence d'arbitrage d'un cycle. La figure 3.5 illustre cette étape (l'étape 2) en 4 sous étapes.

Pour résumer, les requêtes sur le bus qui surviennent au cours du même cycle d'horloge sont classées en ordre de priorité. Elles sont alors placées dans une liste qui est conservée de cycle en cycle, et les requêtes sont élues en ordre FIFO à ce moment.

3. L'étape suivante consiste à notifier à l'arbitre que la transaction a commencé. Ceci permet de verrouiller le canal pendant le transfert (modélisation d'un accès exclusif).
4. Il faut alors simuler la latence de transfert. La latence dans le bus est calculée de la même façon que dans le cas du `space_channel_xbar`.
5. Enfin, le message est transmis dans l'adaptateur cible et l'arbitre est informé que le transfert est terminé.

3.3.8 Composant null_device

Le `null_device` est simplement un périphérique qui procure une implémentation à la classe `space_base_device`, donc qui possède une interface `space_device_if`, mais sans toutefois fournir aucune fonctionnalité. Il peut être utilisé sur le canal `glue_channel`, `space_channel_xbar` ou `space_channel_bus` lorsque le concepteur n'a pas besoin de périphérique dans son architecure matérielle. Les canaux doivent avoir au minimum un périphérique de connecté, le `null_device` sert à assurer cette présence minimale.

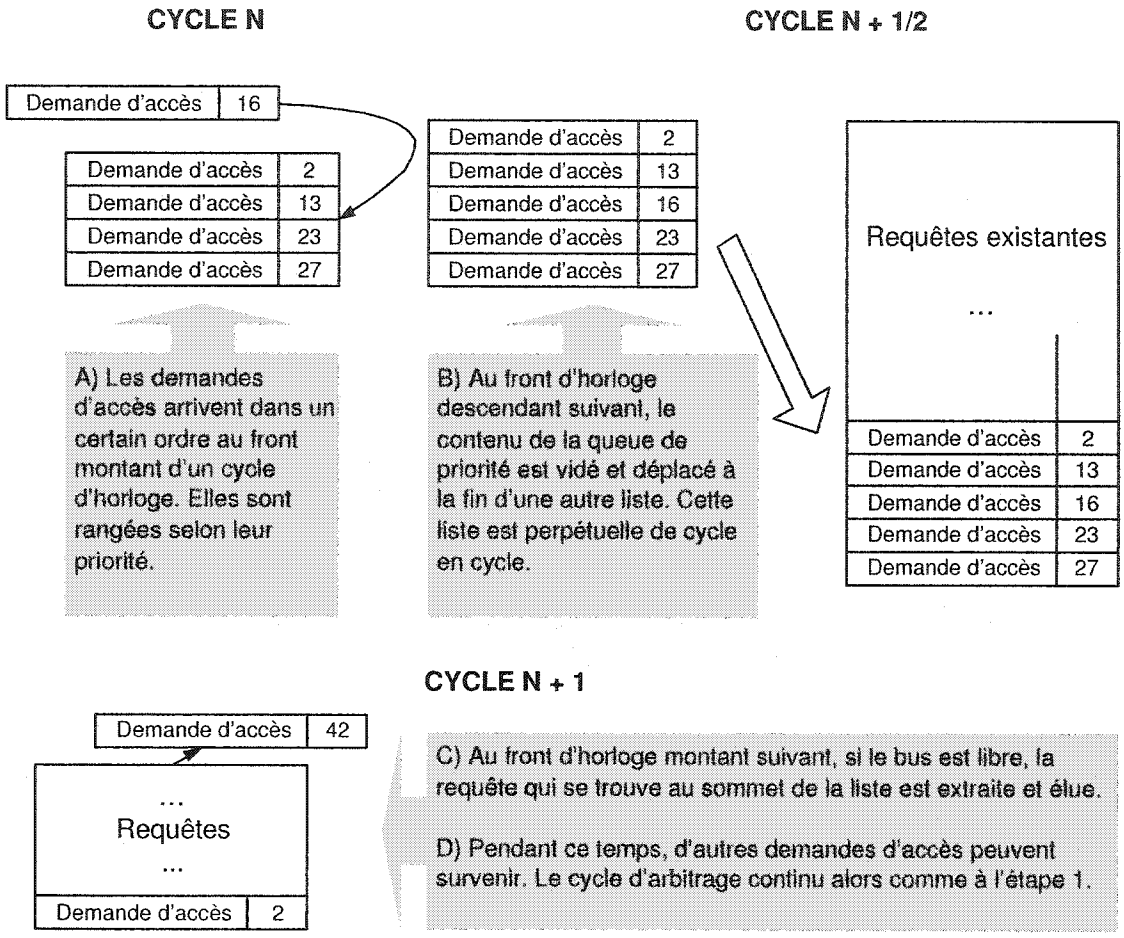

FIGURE 3.5 Algorithme d'arbitrage du space_channel_bus

3.4 Support logiciel sur la plate-forme

Cette section présente les périphériques utilisés pour la simulation du logiciel dans SPACE. Il est à noter que le modèle du processeur, le décodeur ainsi que les mémoires ont été conçus et implémentés par un autre membre de l'équipe [9], c'est pourquoi nous n'allons pas les détailler.

3.4.1 Composant irq_manager

Le rôle du gestionnaire d'interruptions est de permettre à plusieurs périphériques de profiter du mécanisme d'interruption du processeur, même si ce dernier n'a habituellement qu'une seule broche dédiée aux interruptions. Certains processeurs, le ARM par exemple, sont munis de deux broches de la sorte : nIRQ (interruption) et nFIQ (interruption rapide). Dans ce cas, si seulement deux périphériques utilisent les interruptions dans l'application, il est possible de se passer d'un gestionnaire d'interruptions. Dans tous les autres cas, il faut un module qui fait la gestion de tous les signaux d'interruption des périphériques.

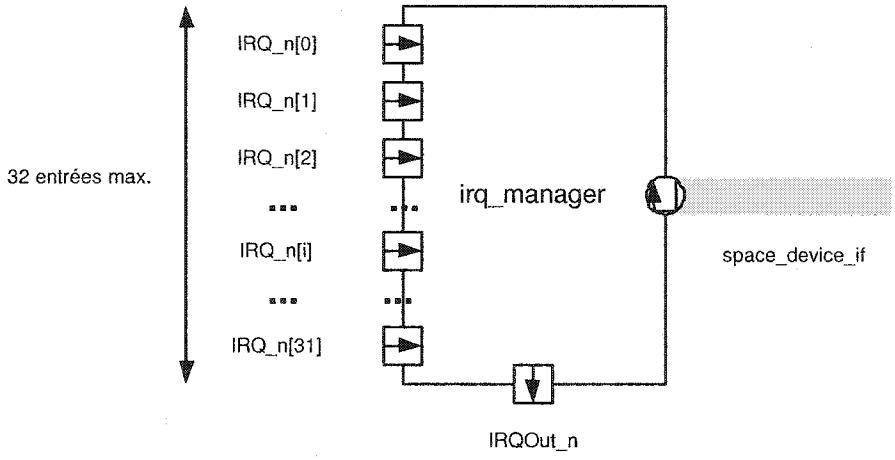

FIGURE 3.6 Schéma bloc du gestionnaire d'interruptions

La figure 3.6 illustre bien la structure du périphérique. Nous avons opté pour une architecture des plus simples. Le gestionnaire est asynchrone et est composé essentiellement d'une méthode de SystemC (`SC_METHOD`) qui est sensible aux signaux d'entrées. Le nombre d'entrées peut varier entre 1 et 32, car le registre d'interruptions interne doit tenir sur 32 bits étant donné qu'il est représenté par une variable entière. Les entrées et le port de sortie pour l'interruption sont tous actifs bas, ce qui permet de mettre en cascade plusieurs gestionnaires d'interruptions. Le

nombre total de lignes d'interruptions n'est donc pas limité à 32 ; de toute façon, il est difficile d'imaginer un cas où plus d'un gestionnaire serait requis.

Le bit le moins significatif du registre d'interruption (i.e. le bit 0) est associé au périphérique qui émet l'IRQ 0, le bit 1 est associé à l'IRQ 1 et ainsi de suite. Le registre ne représente pas l'état actuel des signaux d'interruptions, mais contient plutôt de l'information au sujet des IRQ qui ont été activées depuis un certain moment. Il s'agit donc d'une mémoire qui retient toute activité des signaux d'interruption. Lorsqu'un signal IRQ est activé, c'est-à-dire que le signal passe à l'état " faux " (logique négative), le bit correspondant à cet IRQ est mis à 1 dans le registre d'interruption. Si le signal d'IRQ passe à " vrai ", le bit du registre d'interruption demeure à 1.

Il est possible d'accéder au `irq_manager` en lecture, comme s'il s'agissait d'une mémoire de 32 bits. Dans ce cas le gestionnaire d'interruption retourne son registre d'état. Lorsqu'une écriture est faite à l'adresse du gestionnaire, celui-ci remplace la valeur de son registre d'interruption par la nouvelle valeur donnée du mot de 32 bits. À tout moment, lorsque l'état du registre d'interruption change, le gestionnaire ajuste la valeur de son port de sortie. Il place la valeur "fausse" si au moins 1 des bits du registre d'interruption est à 1 et la valeur "vraie" si le registre d'interruption ne contient que des zéros (logique négative).

Voici un exemple d'enchaînement des opérations qui pourrait être réalisé pour programmer et utiliser le gestionnaire d'interruption depuis la partie logicielle de SPACE. Nous supposons que le système d'exploitation connaît l'adresse du gestionnaire d'interruption.

1. Le système d'exploitation initialise le gestionnaire d'interruption en lui écrivant un entier de 32 bits qui a la valeur nulle (0x00000000).

2. À un moment donné, un périphérique émet une interruption. Attention, le périphérique doit utiliser la logique négative pour émettre son interruption.
3. Le gestionnaire reçoit l'interruption et met à jour son registre.
4. En mettant à jour son registre, le gestionnaire constate qu'au moins 1 des bits est activé, par conséquent, il active (met à 0) la broche d'interruption de sortie (`IRQOut_n`).
5. Quand le processeur sera à l'écoute des interruptions (car il peut les masquer), il verra que le gestionnaire d'interruptions lui envoie un signal.
6. Le système d'exploitation entre dans la routine de traitement des interruptions (ISR).
7. Dans l'ISR, une des premières choses qui devraient être faite est de désactiver les interruptions. Normalement le processeur désactive les interruptions lui-même avant de commencer l'exécution de la routine.
8. Connaissant l'adresse du gestionnaire d'interruption, le système d'exploitation fait une lecture de 32 bits à cette adresse.
9. Le gestionnaire d'interruption lui répond et lui envoie le contenu de son registre.
10. Le système d'exploitation analyse la valeur reçue et choisie le numéro d'IRQ qui sera traité, par un algorithme qui peut implémenter n'importe quelle politique d'arbitrage.
11. Dans certains cas, il faut faire une lecture ou une écriture à l'adresse du périphérique que l'on a choisi de traiter, pour lui indiquer un accusé de réception (ACK). Cela dépend du fonctionnement du périphérique.
12. Le RTOS doit mettre le bit correspondant à l'IRQ qui sera traitée à zéro dans la variable de 32 bits.
13. Cette valeur doit ensuite être écrite à l'adresse du gestionnaire.

14. Ce dernier mettra à jour son registre à la nouvelle valeur reçue et ajustera la valeur de la broche d'interruption de sortie en conséquence. Il est à noter que même si le périphérique maintient à "faux" la valeur du signal d'interruption, le gestionnaire n'en voit rien, car il doit y avoir un changement d'état sur le signal pour que le gestionnaire d'interruption mette à jour son registre.
15. Le système d'exploitation traite la demande du périphérique qui a produit l'interruption choisie. C'est à ce moment que le périphérique risque de remettre à "vraie" la valeur de sa broche d'interruption, pour signaler qu'on lui a répondu (à moins que cela ne soit pas déjà fait, à l'étape de l'ACK).
16. Le système d'exploitation termine son traitement (ISR) et réactive la sensibilité aux interruptions.
17. La boucle recommence à l'étape 2.

3.4.2 Composant timer

La décision d'utiliser un RTOS sur la plate-forme implique que nous avons besoin d'une minuterie pour produire un signal d'horloge temps réel. La minuterie est un périphérique de la plate-forme, par conséquent elle est branchée sur le canal et il est possible de l'accéder comme une mémoire. Elle possède un port d'entrée pour l'horloge ainsi qu'un port de sortie pour produire son interruption (un signal qui est actif bas et donc directement compatible avec le gestionnaire d'interruptions).

Lors de l'instanciation de la minuterie, le concepteur peut spécifier son décompte initial, i.e. le nombre de départ pour le compte à rebours. Ensuite, en cours de simulation, une lecture retourne le décompte interne actuel de la minuterie et une écriture permet d'envoyer des commandes à la minuterie. Il faut absolument utiliser une structure de données prédéfinie pour envoyer des commandes. Cette structure comporte un champ pour une commande (un nombre entier) et un champ pour une

valeur (aussi un nombre entier) et est envoyée à la minuterie à une seule adresse.

Les commandes possibles sont :

TIMER_RESTART Un redémarrage de la minuterie ;

TIMER_LOAD Le chargement d'une nouvelle valeur pour le décompte initial ;

TIMER_TOGGLE_ENABLE L'arrêt ou le départ du compte à rebours. Cela peut servir à arrêter puis repartir la minuterie quelques instants plus tard, tout en conservant le même décompte.

Le champ "valeur" de la structure n'est donc pertinent que lorsque la commande est un chargement. Le chargement d'une nouvelle valeur initiale engendre automatiquement un redémarrage de la minuterie. Il est aussi possible de demander à la minuterie de redémarrer sans changer la valeur actuelle de son décompte initial. Pour ce faire, il faut utiliser la commande `TIMER_RESTART`. La troisième commande, `TIMER_TOGGLE_ENABLE`, active ou désactive la minuterie.

La minuterie contient un seul processus synchrone (`SC_CTHREAD`). Si elle est activée, le compte interne de la minuterie est décrémenté à chaque cycle d'horloge, sinon le décompte reste constant. Lorsque le décompte atteint zéro, le signal d'interruption est levé et rebaissé au cycle suivant. Après être arrivé à zéro, le décompte est remis au compte initial automatiquement.

Pour utiliser la minuterie depuis le côté logiciel, nous supposons que le système d'exploitation connaît son adresse. De plus, la minuterie doit être branchée sur le canal, sur le port des périphériques. Ceci permet aux autres modules de communiquer avec elle. De plus, la broche de sortie pour l'interruption devrait être branchée dans le gestionnaire d'interruption. Les quelques étapes suivantes décrivent la façon typique d'utiliser la minuterie depuis le côté logiciel.

1. Au départ, la minuterie est inactivée. Il est mieux de laisser le système d'exploitation démarrer avant d'activer la minuterie.
2. Ensuite, la première chose à faire est de configurer la minuterie. Pour ce faire, il faut lui envoyer une commande de chargement (`TIMER_LOAD`) et de spécifier le nombre de cycle d'horloge pour le décompte.
3. Ensuite il faut activer la minuterie. Pour ce faire, il suffit d'envoyer la commande d'activation (`TIMER_TOGGLE_ENABLE`). La minuterie se met à décompter et émet une interruption à la fin de son cycle, puis recommence à décompter.

3.4.3 Composant device_adapter

Lorsque le processeur effectue des opérations sur la mémoire, les appels sont redirigés vers le décodeur. Le décodeur examine l'adresse et retransmet l'appel vers le bon périphérique : celui pour qui la plage d'adresse contient l'adresse de destination de l'accès mémoire. Un de ces périphériques peut être le `device_adapter`. Son rôle est de retransmettre les requêtes d'entrées/sorties `mem_read()` et `mem_write()` sur le canal TF. Ainsi, le `device_adapter` implémente l'interface du processeur et du décodeur (`decode_data_if`) et possède un port de type `space_channel_if`.

Du point de vue du processeur, le `device_adapter` possède une certaine plage d'adresse visible. Cette plage d'adresse définie la "fenêtre" ou plage mémoire que pourra accéder le processeur concernant les périphériques sur le canal TF. La figure 3.7 illustre le concept. Sur cette figure, nous voyons pourquoi l'adaptateur de périphérique doit avoir une plage d'adresse qui contient les trois plages d'adresses des périphériques sur le canal, sans quoi le processeur ne pourra pas les accéder, en passant par le décodeur.

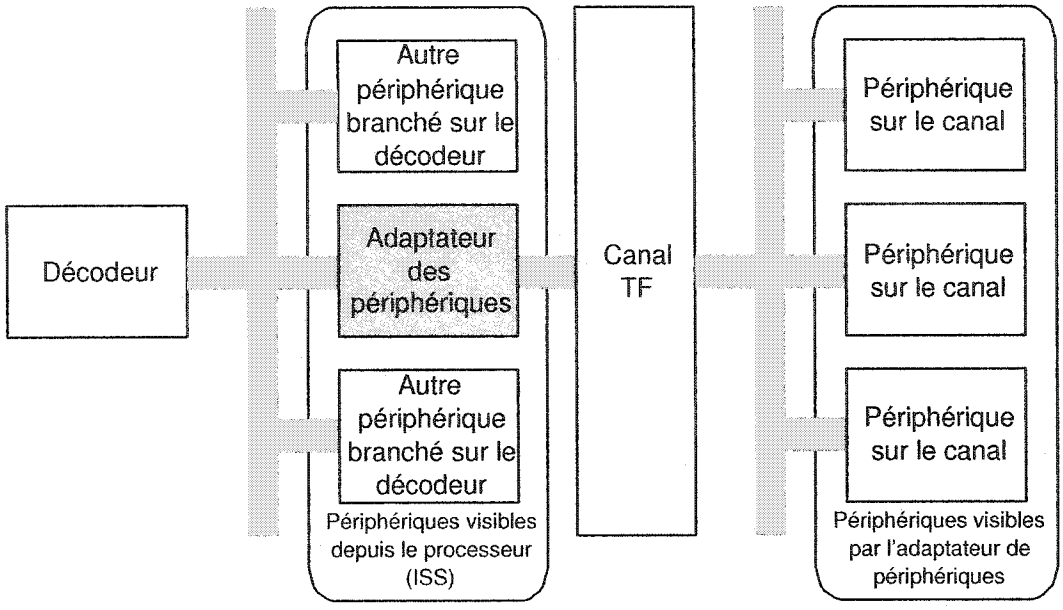

FIGURE 3.7 Domaines d'adresses pour le processeur et le device_adapter

3.4.4 Composant iss_adapter

Sur la figure 3.2, nous voyons qu'un autre périphérique fait le pont entre le décodeur et le canal : l'`iss_adapter`. C'est ce périphérique qui s'occupe principalement de la communication entre les modules logiciels et matériels de l'usager. L'adaptateur du processeur possède les deux interfaces, celle qui permet de brancher le décodeur et celle qui permet de brancher le canal. De plus, un port vers le canal lui permet d'initier des transactions `write()` et `nb_write()`, tout comme les adaptateurs de modules. Finalement, il est équipé d'un port d'entrée pour l'horloge et d'un port de sortie pour émettre une interruption.

Les événements externes qui peuvent survenir pour stimuler l'`iss_adapter` sont les suivantes : une lecture ou une écriture de la part du processeur et une écriture bloquante ou non bloquante de la part du canal. Les lectures depuis le canal sont improbables, car selon notre conception il n'y a que les opérations d'écriture qui voyagent

par le canal TF. De plus, une autre restriction nous a simplifié l'implémentation de ce périphérique : il n'y a que les accès mémoire de 32 bits qui peuvent être dirigés vers l'`iss_adapter` à partir du processeur. Les autres accès de 8 bits et de 16 bits sont considérés invalides.

La figure 3.8 présente un aperçu du fonctionnement de l'`iss_adapter`.

Lorsqu'un module matériel de l'usager désire envoyer un message à un module logiciel, son message passe par le canal et aboutit dans l'`iss_adapter`. S'il s'agit d'une écriture non bloquante, l'adaptateur crée une requête et l'insère dans une liste (`m_RequestsFromChannel`). S'il s'agit d'une écriture bloquante, on l'insère aussi dans la liste, mais en plus on doit attendre que le message soit transmis au côté logiciel avant de continuer.

Un processus nommé `decoder_side_thread()`, un `SC_CTHREAD`, s'occupe de transmettre ces requêtes. Le processus commence par vérifier s'il y a quelque chose à transmettre et si oui, on récupère le premier message et on le place dans un tampon d'envoie. On ajoute une entête au message à envoyer dans le tampon, pour informer le côté logiciel de l'identité de l'émetteur, de l'identité du destinataire ainsi que de la taille du message. Lorsque le tampon est prêt à être envoyer, l'`iss_adapter` place à "faux" (logique négative) sa broche d'interruption. Le processus se place alors en attente d'un événement qui consiste en la fin de la transmission. Lors de l'arrivée de cet événement, on supprime le tampon rendu inutile et le processus recommence sa boucle.

Du côté logiciel, la communication provenant du matériel débute avec l'interruption. Suite à un certain traitement logiciel en cas d'interruption que nous ne détaillons pas ici [42], le logiciel veut maintenant venir chercher le message qui lui est destiné. À ce moment, l'ISS effectue une série de lectures en mémoire à l'adresse

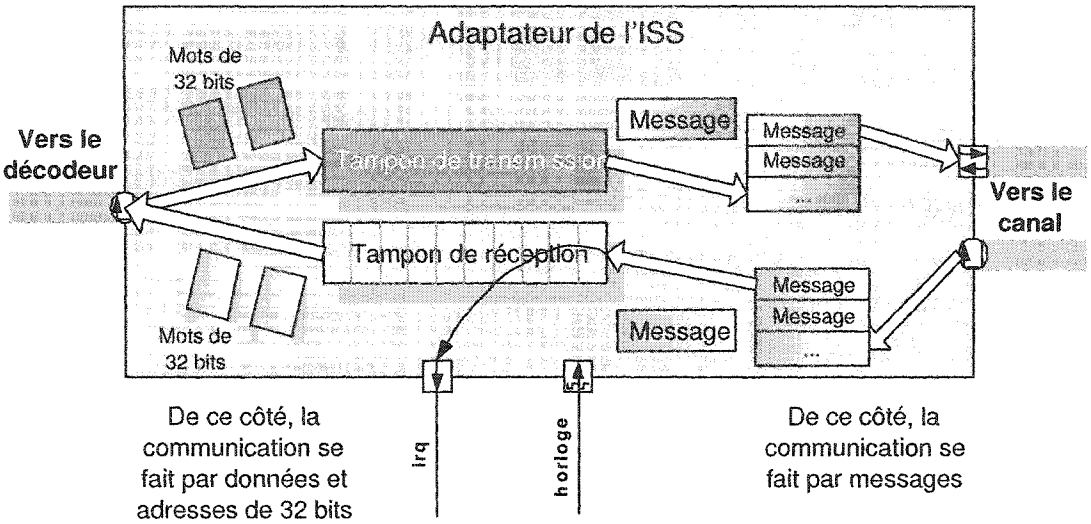

FIGURE 3.8 Fonctionnement du périphérique iss_adapter

de l'`iss_adapter`. Lorsque ce dernier reçoit la première requête de lecture, il comprend que le logiciel a reçu l'interruption et donc l'`iss_adapter` la désactive. Le logiciel lit alors 32 bits par 32 bits le tampon qui contient l'entête et le message. Lorsque le tampon est entièrement lu, le processus `decoder_side_thread` en est informé par un événement.

La communication dans l'autre direction, c'est-à-dire du logiciel vers le matériel, est un peu plus simple et ne nécessite pas l'utilisation des interruptions. Les requêtes d'écriture sont décomposées en blocs de 32 bits du côté logiciel, simplement parce que le processeur a un bus de données de cette largeur. Les écritures passent ensuite par le décodeur et arrivent dans l'`iss_adapter`. Encore une fois, l'ajout d'un entête au message permet à l'adaptateur de savoir à l'avance combien de mots de 32 bits lui seront transmis. Un tampon mémoire de la bonne taille peut ainsi être créé et utilisé pour recevoir les données pendant plusieurs cycles. Quand le message est reçu au complet, une requête est créée et stockée dans une liste (`m_RequestsFromDecoder`). Le tampon mémoire peut ensuite être effacé.

Un autre processus `SC_CTHREAD` est utilisé dans l'`iss_adapter`, cette fois pour envoyer les messages reçus depuis le côté logiciel vers le canal TF. Le processus `channel_side_thread` récupère une requête de la liste `m_RequestsFromDecoder` et l'envoie dès que possible, de façon analogue au processus `outgoingWriteThread()` présent dans les adaptateurs de modules quand ceux-ci sont configurés pour fonctionner avec processus (voir la figure 3.4).

CHAPITRE 4

RÉSULTATS, ANALYSE ET DISCUSSION

4.1 Exemple d'utilisation de la plate-forme

Pour bien illustrer notre approche, afin de montrer les transitions d'un niveau à l'autre dans SPACE, nous jugeons nécessaire de détailler les étapes une après l'autre de notre méthodologie à l'aide d'un petit exemple simple. L'application en question comporte à la fois un flot de données continu, quelques tâches de contrôle et une boucle de rétroaction qui veut simuler un certain asservissement. Nous avons conçu cet exemple pour exprimer la faisabilité d'utiliser notre plate-forme pour une application qui exploite cette diversité de domaines d'application.

La figure 4.1 présente les modules de l'usager qui vont interagir dans l'exemple. Les rôles qu'on leur attribut sont les suivants :

Producteur Il génère des données périodiquement. Pour ce faire, un processus attend sur une condition temporelle. Au moment venu, il envoie une donnée entière (32 bits) au Filtre.

Filtre Ce module effectue une lecture bloquante vers le Producteur, pour attendre la donnée. Lorsqu'elle est reçue, le Filtre la modifie en fonction de trois coefficients entiers qui lui sont propres. Ces coefficients ont une valeur initiale et il est possible de les changer en envoyant une commande spéciale au Filtre.

Sélecteur Le Sélecteur reçoit les données modifiées par le Filtre et les range en mémoire, dans une plage d'adresse définie.

Contrôleur Périodiquement, le Contrôleur se réveille, demande d'abord au Sélecteur de changer de plage mémoire pour ranger les données. Cela laisse les données déjà écrites en mémoire dans un état sûr. Le Contrôleur demande ensuite au module Analyseur d'aller effectuer un calcul sur ces données. L'Analyseur lui retourne un résultat de calcul et ce résultat est utilisé pour calculer des nouveaux coefficients à envoyer au filtre.

Analyseur Comme mentionné, l'Analyseur se met en attente d'une écriture de la part du Contrôleur. Ce dernier lui envoie des informations sur la plage mémoire à analyser.

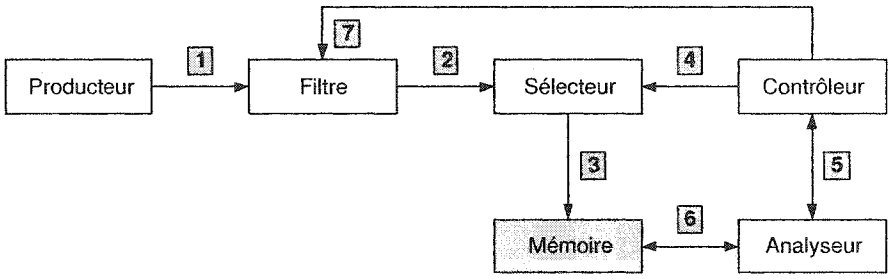

FIGURE 4.1 Schéma de principe de l'exemple

Les numéros sur la figure 4.1 illustrent graphiquement l'ordre des transactions qui ont été exprimées ci haut.

4.1.1 Niveau UTF

La première version exécutable de la spécification consiste en une description en SystemC au niveau UTF. Les modules sont conçus avec les règles architecturales de SPACE et sont connectés au canal `glue_channel`. Comme une mémoire est requise pour le stockage des données, le périphérique `ram_data` peut être utilisé. Les connexions sont simples et directes, comme le montre la figure 4.2.

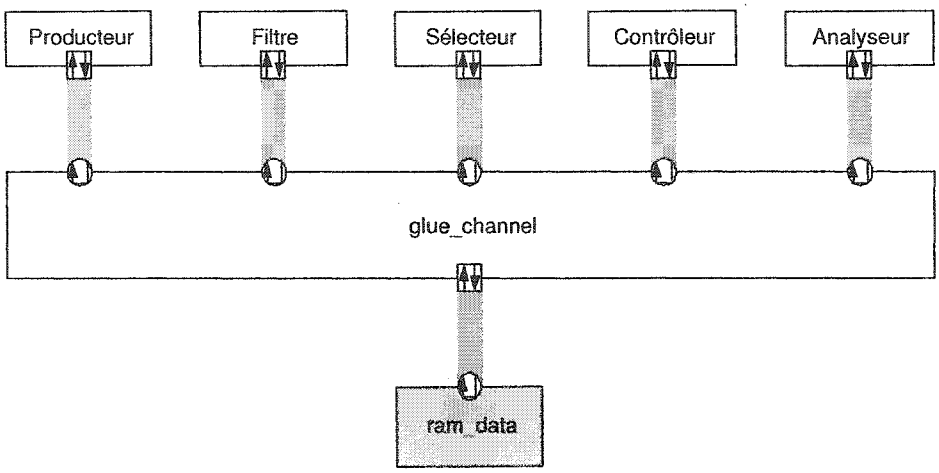

FIGURE 4.2 Exemple avec le Glue Channel

4.1.2 Niveau TF - Crossbar

Une fois que l'application de l'usager fonctionne correctement après l'avoir simulée avec le `glue_channel`, l'étape suivante est de convertir l'application au niveau TF. Le premier canal TF qui devrait être essayé est le réseau point à point (Crossbar, ou X-Bar) parce que c'est le plus simple à utiliser.

La figure 4.3 illustre la nouvelle version de l'exemple, cette fois au niveau TF en utilisant le canal Crossbar. En plus de devoir remplacer le `glue_channel` par le `space_channel_xbar`, l'usager a la responsabilité de remplacer la classe de base des modules, pour utiliser celle qui a été définie pour le niveau TF. Cette modification nous oblige à convertir les processus `SC_THREAD` utilisés au niveau UTF par des `SC_CTHREAD`. Une autre modification consiste à instancier autant d'adaptateurs de modules qu'il y a de modules usager. Dans notre exemple, il nous faut donc 5 adaptateurs de module. L'usager a le choix d'activer ou non les processus à l'intérieur des adaptateurs. Nous avons choisi le cas où les adaptateurs n'ont pas de processus. En plus de la structure des modules et des périphériques, un certain

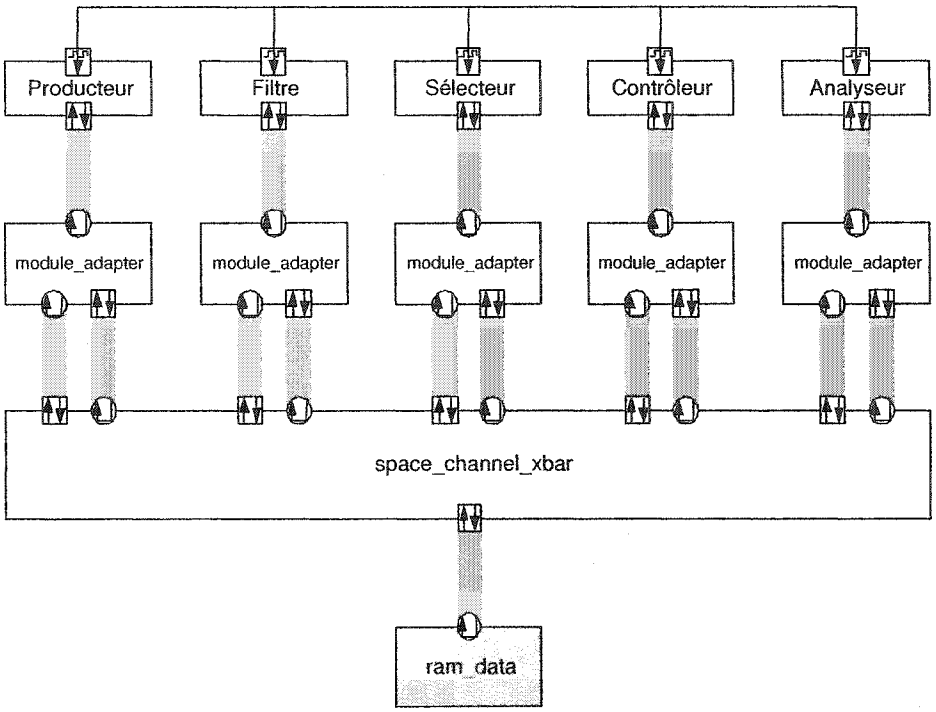

FIGURE 4.3 Exemple avec le Space Channel Crossbar

nombre de connexions doivent être ajustées. Les modules de l'usager ont maintenant un port d'horloge et il faut les connecter à un signal d'horloge global pour toute l'architecture. Les modules de l'usager doivent en plus être branchés sur leurs adaptateurs respectifs. Au niveau du canal, il y a maintenant un multi-port pour les adaptateurs et il faut brancher ce port à toutes les interfaces des adaptateurs. Ces derniers doivent à leur tour être branchés sur le canal. Il est à remarquer que nous avons simplifié le schéma de la figure 4.3 en affichant les multi-ports comme des ensembles de ports distincts. Cette simplification veut seulement ajouter de la clarté au schéma.

4.1.3 Niveau TF - Bus

Un raffinement trivial qui suit la version TF avec le Crossbar de l'exemple est d'utiliser un canal de type Bus.

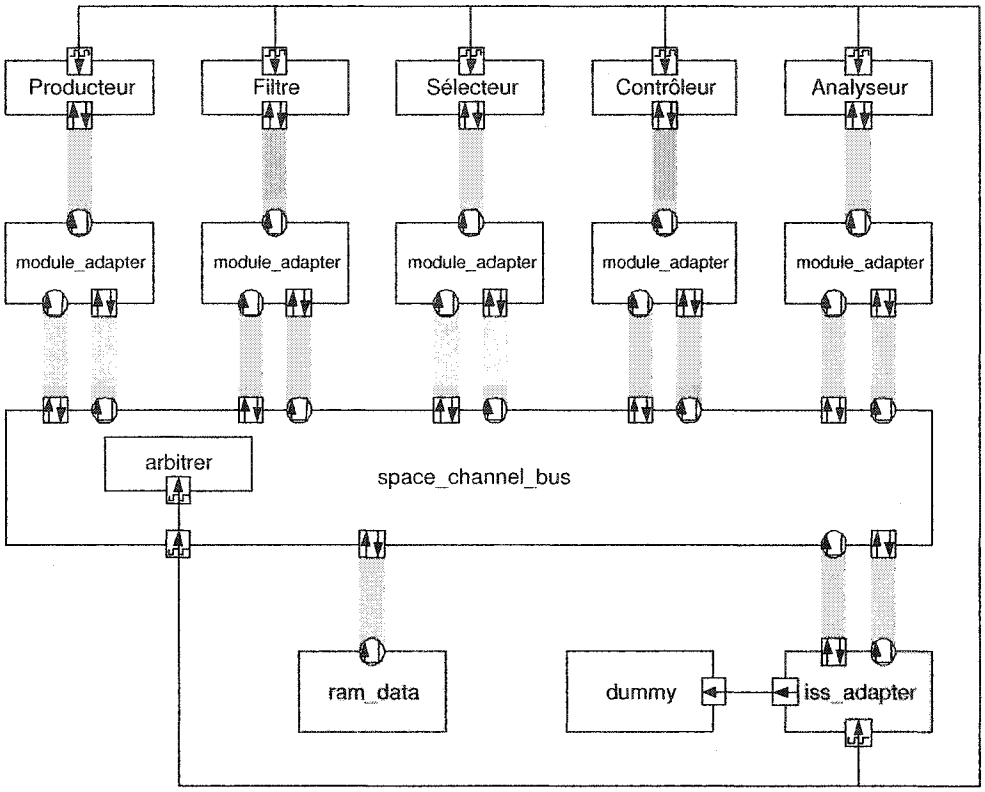

FIGURE 4.4 Exemple avec le Space Channel Bus

La nouvelle version de l'exemple avec le `space_channel_bus` est illustré à la figure 4.4. Un arbitre est présent dans le canal et il possède un `SC_CTHREAD`. Par conséquent, un port d'horloge vient s'ajouter au canal. Il faut le connecter au même signal d'horloge que celui des modules. Le `space_channel_bus` possède un port dédié pour l'`iss_adapter`. Cependant, la version actuelle de l'exemple est encore 100% matérielle et donc nous n'avons pas besoin d'utiliser l'`iss_adapter`. Néanmoins, nous sommes obligés de l'instancier pour respecter les règles architec-

turales de SystemC, qui dictent qu'un port doit toujours avoir un minimum de 1 connexion. L'`iss_adapter` se connecte au canal de la même façon qu'un adaptateur de module, sauf qu'il faut en plus connecter son port d'horloge au signal d'horloge global de l'architecture et brancher aussi un signal booléen à son port d'interruption. Les interruptions ne sont pas utilisées, car nous n'avons pas de processeur, ni de gestionnaire d'interruption. Il faut cependant connecter le signal à un port quelconque. L'utilisation d'un module bidon (nommé "dummy" sur la figure 4.4) règle le problème.

4.1.4 Niveau TF - Partitionné avec le Bus

Jusqu'à maintenant nous avons raffiné l'exemple du niveau UTF jusqu'au niveau TF, mais seulement en conservant les modules en matériel. L'étape subséquente consiste donc à essayer un premier partitionnement logiciel/matériel.

Comment peut-on choisir l'architecture SPACE cible ? Évidemment, la structure architecturale reflète le choix du partitionnement. L'idée est de débuter en faisant un choix d'un certain partitionnement qui semble logique. L'exemple partitionné en deux modules logiciels (Contrôleur et Analyseur) et trois modules matériels (Producteur, Filtre et Sélecteur) qui utilisent le canal de type bus est illustré à la figure 4.5. Nous justifions ce choix de partitionnement en se basant sur le fait que le producteur, le filtre et le sélecteur communiquent en chaîne à chaque fois qu'une donnée est produite. La fin de la chaîne se termine par une écriture en mémoire. Notre idée est donc de les implémenter en matériel. Les modules Contrôleur et Analyseur consistent principalement en unités de contrôle. Nous avons choisi de les implémenter en logiciel afin de profiter du parallélisme logiciel/matériel et aussi parce que les calculs que doive exécuter l'Analyseur sont très simples et ne demanderont donc pas beaucoup de cycles d'exécution de la part du processeur ARM.

Des modifications assez majeures s'imposent pour ajouter une partie logicielle à l'architecture. D'abord, notons l'ajout du processeur (ou plutôt de l'ISS) et des périphériques essentiels pour son utilisation : le décodeur, la mémoire de code, la minuterie, le gestionnaire d'interruptions, l'adaptateur du processeur et l'adaptateur des périphériques. La minuterie et l'ISS doivent être connectés sur l'horloge et les sources d'interruptions (c'est-à-dire la minuterie et l'adaptateur du processeur) doivent être connectées au gestionnaire d'interruption. Le port d'interruption de sortie de ce dernier doit être branché au port d'entrée pour les interruptions sur le processeur. L'ISS a aussi un port d'entrée pour les interruptions rapides (FIQ) et pour la remise à zéro (reset). Nous avons créé un module bidon spécial pour y connecter ces deux ports qui ne sont pas utilisés lors de la simulation. Ce module n'est pas inclus sur la figure 4.5 pour simplifier le schéma. Il faut également remarquer que la mémoire de données est connectée à la fois au canal et au décodeur. Cela permet à la partie matérielle et à la partie logicielle d'utiliser la mémoire.

Nous omettons ici les explications concernant la configuration logicielle pour permettre la simulation complète du système. Ce sujet dépasse le cadre de ce mémoire et le lecteur pourra y trouver de l'information dans [42].

4.2 Outils de mesure de performance

L'utilisateur doit être guidé par des résultats de simulation pour fixer le partitionnement logiciel/matériel ainsi que les autres paramètres de SPACE, comme la taille des mémoires, des piles de tâches en logiciel, etc. Pour ce faire, l'utilisateur peut utiliser des outils inclus dans SystemC ou encore d'autres mécanismes que nous avons conçus.

Les bibliothèques de base du langage C/C++ permettent de calculer le temps

d'exécution d'un programme. Cela peut donner une idée de la performance qu'apporte un certain algorithme ou une certaine optimisation. Cependant, la notion de temps à ce niveau fait référence au temps pour exécuter l'application de l'usager en SystemC et non pas au temps simulé.

En SystemC, il est plus pertinent de considérer une autre dimension temporelle : celle qui est propre à la simulation. Pour obtenir le temps de simulation qui s'est écoulé jusqu'à un certain moment, on peut utiliser les fonctions suivantes [37] :

sc_simulation_time() Cette fonction retourne le temps de simulation actuel dans l'unité de temps définie par défaut en une valeur de type "double" (nombre flottant en double précision sur 64 bits). L'utilisateur peut lui-même définir l'unité de temps par défaut avant la phase d'élaboration du modèle, lors du lancement du programme. Dans SPACE, l'unité de temps par défaut est la nanoseconde.

sc_time_stamp() Il s'agit d'une autre fonction disponible avec SystemC pour obtenir le temps actuel de la simulation. La seule différence avec la fonction `sc_simulation_time()` est que la valeur de retour est de type `sc_time`, une classe de SystemC.

Dans les sous-sections qui vont suivre, nous allons présenter les outils pour récolter des statistiques que nous avons ajoutés aux composants de SPACE pour recueillir des informations sur les performances dynamiques en cours de simulation.

4.2.1 Outil dans le glue_channel

Au niveau UTF, la notion de cycles d'horloge n'est pas importante pour le concepteur. Par conséquent, nous n'avons pas jugé pertinent l'ajout d'estimateurs de per-

formance perfectionnés. Seul un compteur de requêtes a été ajouté dans le but de pouvoir facilement calculer les performances comparatives entre le `glue_channel` et le `space_channel`. À chaque fois qu'une opération reliée à la communication se termine, en d'autres mots l'envoie d'un message ou d'une requête d'entrée/sortie `read()`, `write()`, `nb_read()`, `nb_write()`, `mem_read()` ou `mem_write()`, une variable membre dans la classe du canal est incrémentée.

4.2.2 Outil dans le space_channel

Dans le cas des canaux TF, soit le Bus et le Crossbar, nous avons implémenté des mécanismes différents de récolte de statistiques pour refléter des modèles différents.

Pour le canal `space_channel_xbar`, il y a deux méthodes disponibles :

getBusyCycles() Nombre de cycles d'horloge total pour lequel le canal était occupé à envoyer des messages.

getNumReq() Il s'agit du même mécanisme que dans le cas du `glue_channel` pour récolter le nombre de transactions effectuées en simulation.

Dans le cas du Bus, la méthode `getBusyCycles()` retourne le nombre de cycles pour lequel le canal était occupé, c'est-à-dire soit qu'il était en train de transférer un message ou soit qu'il était en train de procéder au cycle d'arbitrage. Cette méthode permet donc de fournir des informations sur le pourcentage d'utilisation du Bus.

En plus d'un compteur de requêtes, au cours de la simulation, des informations statistiques sont collectées à l'aide d'un objet nommé moniteur. Le moniteur fournit un rapport dans la console. Les informations qui y sont données sont fournies par les méthodes suivantes :

```
double getWaitingRequestsAvr(void);
unsigned long getWaitingRequestsMax(void);
unsigned long getWaitingRequestsCount(void);
double getNbWaitingCyclesAvr(void);
unsigned long getNbWaitingCyclesMax(void);
unsigned long getNbWaitingCyclesCount(void);
```

Les noms des méthodes sont formés par des combinaisons de 2 préfixes et de 3 suffixes différents. Les préfixes indiquent le type d'information qui est donné :

WaitingRequests Il s'agit du nombre de processus en attente de l'arbitre. À chaque cycle d'horloge (au front montant), le bus connaît le nombre de processus `SC_CTHREAD` qui sont en attente et ce nombre est envoyé au moniteur.

NbWaitingCycles Pour chaque requête, le nombre total de cycles d'horloge d'attente après l'arbitre est envoyé au moniteur. Cela donne donc un indice sur l'importance des situations de contention sur le canal.

Les suffixes Count, Avr et Max symbolisent respectivement le nombre d'échantillons pour calculer la statistique, la moyenne et le maximum parmi les échantillons.

4.2.3 Outil dans le module_adapter

Le périphérique `module_adapter` possède aussi un moniteur intégré ; toutefois l'utilisateur a le choix d'instancier ou non le moniteur au début de la simulation. Lorsque le destructeur est appelé, l'adaptateur vérifier si le moniteur existe et si oui, lui demande de fournir un rapport. Ce rapport statistique est obtenu par l'appel des méthodes suivantes :

```
virtual double getWaitingRequestsAvr(void);
virtual unsigned long getWaitingRequestsMax(void);
virtual unsigned long getWaitingRequestsCount(void);
virtual double getNbWaitingCyclesAvr(void);
virtual unsigned long getNbWaitingCyclesMax(void);
virtual unsigned long getNbWaitingCyclesCount(void);
virtual double getMemoryUsageAvr(void);
virtual unsigned long getMemoryUsageMax(void);
virtual unsigned long getMemoryUsageCount(void);
```

Ces méthodes donnent les renseignements que nous décrivons ici :

WaitingRequests Il s'agit du nombre de requêtes provenant du canal qui sont en attente de réponse dans l'adaptateur. À chaque cycle, le nombre de requêtes en attente est envoyé au moniteur.

NbWaitingCycles Cette mesure est le temps d'attente total d'une requête dans un adaptateur, exprimé en cycles d'horloge.

MemoryUsage À l'intérieur du `module_adapter`, les messages qui arrivent du canal sont stockés dans une liste. Pour chaque nouvelle requête, la taille du message est envoyée au moniteur, ce qui donne une approximation de la mémoire requise pour contenir les requêtes.

Les suffixes ci-dessus ont les mêmes significations que pour le cas du canal (voir `space_channel`). Il est à noter que chaque adaptateur fournit un rapport qui lui est propre, donnant ainsi plus de détails au concepteur. Finalement, les outils de mesure qui sont actuellement implémentés supposent que le concepteur utilise la version du `module_adapter` sans processus.

4.3 Présentation des résultats

Cette section est dédiée à la présentation de résultats concernant des tests de performance avec SPACE. Tous les tableaux de résultats qui ont permis de produire les graphiques que nous allons présenter se trouvent à l'annexe II.

4.3.1 Comparaison entre SPACE et d'autres modèles existants

Nous avons tenté de comparer les canaux de SPACE avec d'autres modèles de canaux existants. Nous avons choisi les deux plus pertinents, soit SOCP [40] et Simple Bus [20]. Le code source de StepNP a été obtenu grâce à une entente entre le Groupe de Recherche en Microélectronique de l'École Polytechnique de Montréal et la division System-on-Chip Platform Automation Group chez STMicroelectronics, localisée à Ottawa. Le code source de Simple Bus fait partie des exemples fournis avec la bibliothèque SystemC 2.0.1 [20] que tous peuvent télécharger à même le site Web.

Les tableaux II.1 et II.2 en annexe donnent les résultats expérimentaux issus de l'exécution de plusieurs bancs d'essais réalisés avec SPACE, SOCP de StepNP et Simple Bus. Notre objectif était de partir de deux exemples donnés avec SOCP et de reconstruire des bancs d'essais similaires avec SPACE et Simple Bus. Le but était de laisser le code de SOCP intact.

L'application utilisée est celle de 8 maîtres et 8 esclaves. La version originale est le programme `funcTest` originalement conçu pour évaluer la performance du canal SOCP fonctionnel. Les maîtres exécutent un processus en continue qui contient deux boucles finies qui s'exécutent l'une après l'autre. D'abord, la première boucle consiste en l'écriture en mémoire (les esclaves sont en réalité des modèles de

mémoires) de données entières du début jusqu'à la fin d'une plage mémoire d'une certaine taille. Ensuite, l'autre boucle consiste simplement en des lectures sur toute la plage mémoire. Le maître vérifie que les données qui y sont lues correspondent à celles qui avaient précédemment été écrites.

Dans SPACE, les maîtres ont été implémentés comme des modules de l'usager et les esclaves sont des mémoires de données. Les modules de l'usager utilisent les méthodes bloquantes `mem_read()` et `mem_write()` pour communiquer. Avec Simple Bus, les écritures et lectures sont effectuées avec les méthodes `burst_write` et `burst_read` et les esclaves sont des instances de la classe `simple_bus_fast_mem`, très similaire à notre mémoire `ram_data`.

Voici maintenant une courte description des différentes versions de bancs d'essais utilisés pour produire les résultats qui ont été regroupés sur le graphique de la figure 4.6. Sur ce graphique, k-op./sec est un raccourci pour milliers d'opérations d'entrée/sortie par seconde.

PT1T 8 maîtres et 8 esclaves connectés sur le `space_channel_xbar`, les adaptateurs de modules ont des processus.

PT1NT 8 maîtres et 8 esclaves connectés sur le `space_channel_xbar`, les adaptateurs de modules n'ont pas de processus.

PT2T 8 maîtres et 8 esclaves connectés sur le `space_channel_bus`, les adaptateurs de modules ont des processus.

PT2NT 8 maîtres et 8 esclaves connectés sur le `space_channel_bus`, les adaptateurs de modules n'ont pas de processus.

PT3 8 maîtres et 8 esclaves connectés sur le `glue_channel`, il n'y a pas d'adaptateurs de modules.

PT4 8 maîtres et 8 esclaves connectés au canal Simple Bus, l'interface bloquante

est utilisée. Cette version n'est pas compatible avec SPACE ; elle a été écrite pour être directement compatible avec Simple Bus.

PT5 8 maîtres et 8 esclaves connectés au canal SOCP fonctionnel. Il s'agit de la version originale du banc d'essai nommé `funcTest`.

PT6 1 seul maître et 1 seul esclave connectés ensemble sans canal. Les ports d'un module (maître ou esclave) sont directement connectés aux interfaces de l'autre module (esclave ou maître). Ce programme nommé `nullModem` était fourni avec la distribution de StepNP que nous avons reçu. Nous aurions pu la modifier pour qu'elle comporte 8 maîtres et 8 esclaves ; cependant nous aurions modifié StepNP et cela allait à l'encontre de nos contraintes imposées.

Comme nous pouvons le remarquer, deux machines différentes ont été utilisées pour les tests. La première est un Sun Blade 100 avec Solaris et CDE d'installés. La deuxième machine est un PC équipé d'un processeur Intel Pentium III cadencé à 667 MHz avec Microsoft Windows 2000 Professional SP4.

Il est à noter que le code de StepNP (et par conséquent tout ce qui concerne SOCP) n'est pas compatible avec les systèmes d'exploitation Windows, c'est pourquoi nous avons d'abord fait des tests sur une machine de type Unix. Les résultats sous Windows n'incluent donc pas les tests PT5 et PT6. Nous avons tout de même jugé pertinent de présenter les résultats sur PC, car plus loin dans ce chapitre nous présenterons d'autres résultats de performance sur cette architecture. Le lecteur intéressé pourra ainsi comparer les différents résultats qui ont été obtenus sur le même poste de travail.

Comme le montre la figure 4.6, le canal le plus rapide est le `glue_channel` et le plus lent, le `space_channel_bus`. Cette dernière version, avec son algorithme d'arbitrage qui implique les deux fronts de l'horloge, les adaptateurs de modules et les processus synchrones `SC_CTHREAD`, est plus lente que la version UTF de plus

de deux ordres de grandeur. Le `glue_channel` est d'ailleurs même plus rapide que la version `nullModem` de SOCP, même si cette dernière ne comporte aucun canal et seulement 1 maître et 1 esclave. Un autre fait intéressant à noter est que les adaptateurs de module peuvent contenir des processus ou non sans faire varier significativement les performances. Notre version avec le Crossbar est à toute fin pratique assez équivalente en performance avec l'implémentation simpliste et incomplète ("simple functional implementation") de SOCP (PT5 versus PT1T et PT1NT). En fait, les deux modèles sont très semblables à propos de leurs niveaux d'abstraction, de leurs caractéristiques et de leur modèle.

4.3.2 Tests paramétriques de performance

En plus de comparer les canaux de SPACE avec d'autres modèles de communication existants, nous voulions déterminer le comportement en simulation de notre implémentation face à des variations sur les paramètres suivants : le type de canal utilisé, le type d'application qui est implémentée par l'usager, le nombre de modules présents et finalement la taille moyenne des messages qui transigent dans les canaux de communication.

Une brève description de ces paramètres s'impose.

Type de canal Pour chaque application, nous avons conçu une version différente qui utilise un canal différent, le `Glue Channel`, `Crossbar` et `Bus`. Il est de coutume d'utiliser le mot X-bar à la place de Crossbar.

Types d'applications Nous voulions savoir comment réagit SPACE face à des applications totalement différentes. Nous avons donc conçu les trois applications suivantes :

Ring Dans cette application, tous les modules de l'usager envoient un mes-

sage à leur module voisin et reçoivent un message de leur autre voisin. Les voisins sont déterminés par les ID des modules. Pour éviter les interblocages, le dernier module qui est créé est différent des autres : il débute par une lecture ou lieu d'une écriture.

War Cette application nécessite que le nombre de modules soit pair, car il s'agit simplement d'une application producteur/consommateur. Chaque module producteur envoie à répétition un message à un consommateur désigné. Lors de l'instanciation des modules de l'usager, des paramètres dans les constructeurs permettent de déterminer quel producteur va envoyer des messages à quel consommateur.

Stress Tous les modules de l'application Stress sauf 1 envoient à répétition un message à un même module, qui ne fait que lire l'un après l'autre les messages reçus. Pour N modules, il y a ainsi N-1 producteurs et 1 consommateur.

Nombre de modules Dans tous les cas, pour toutes les applications, aucun périphérique n'est utilisé, il n'y a seulement que des modules de l'usager. Comme les canaux nécessitent qu'au moins un périphérique soit connecté, nous avons utilisé le `null_device`.

Taille des messages Pour toutes les applications, les modules s'échangent des messages. Nous avons rendu paramétrique la taille de ces messages, qui est exprimée en nombre d'entier de 32 bits qui le composent. Par exemple, une taille de 10 signifie que la taille du message est de 10 fois 32 bits, soit 40 octets.

Nous avons conçu neuf programmes différents, soit la combinaison des trois applications possibles avec les trois types de canaux possibles. Chaque programme prend trois arguments en paramètre : le nombre de modules, la taille des messages et une variable qui sert à contrôler l'affichage (ON ou OFF) en cours de simulation.

Évidemment, nous avons choisi l'option de ne pas rien afficher en cours de simulation pour les fins de l'expérience, car l'affichage d'informations de déverminage aurait trompé les résultats de simulation.

Les tableaux de résultat II.3 II.4 et II.5 à l'annexe II contiennent les résultats numériques qui ont servi à tracer les histogrammes des figures 4.7, 4.8 et 4.9. Dans tous les cas, la machine de test était un Pentium III à 667 MHz.

Le premier cas qui est présenté à la figure 4.7 illustre les performances de SPACE en milliers d'opérations par seconde en fonction du type de canal et du type d'application. Le nombre de modules était constant à 100 et la taille des messages était de 1 entier. La première constatation est que le type d'application semble influencer davantage les performances au niveau TF qu'au niveau UTF. Il est intéressant de noter que les performances de War avec le `glue_channel` et le `space_channel_xbar` sont dans le même ordre de grandeur. Également, les applications Ring et Stress semblent donner les même performances pour un canal donné.

Pour la deuxième série de tests paramétriques que nous avons effectués, nous avons gardé constant le type d'application à War et la taille des messages était encore constante à 1 entier. Nous avons testé les trois types de canaux et fait varier le nombre de modules participants. Outre le fait que les performances diminuent en fonction du nombre de modules, il y a deux remarques intéressantes à souligner au sujet des résultats de la figure 4.8. D'abord, nous pouvons remarquer que les performances comparatives entre le canal Bus et les deux autres canaux s'éloignent de plus en plus au fur et à mesure que le nombre de modules augmente. Ensuite, nous pouvons également constater que les performances du `space_channel_xbar` deviennent meilleures que celles du `glue_channel` lorsque le nombre de modules augmente significativement (quelque part entre 100 et 500 modules). Pour expliquer ce phénomène étrange, nous avons une hypothèse basée sur le fonctionnement

interne. En effet, le `glue_channel` contient une liste STL qui est utilisée pour stocker les requêtes de tous les modules et la recherche se fait linéairement. Le `space_channel_xbar` ne contient pas de liste, les messages sont plutôt stockés dans les adaptateurs de module. Comme il y a un seul adaptateur pour chaque module, au lieu d'avoir une longue liste dans le canal, nous avons plusieurs petites listes de requêtes locales dans chacun des adaptateurs. Lorsqu'un module effectue une requête de lecture, le temps de recherche est par conséquent plus court.

Le cas #3 est semblable au cas #2, sauf qu'au lieu de faire varier le nombre de modules, nous l'avons fixé à 10 et fait plutôt varier la taille des messages. L'application utilisée est War. Les résultats de simulation sont présentés à la figure 4.9. Nous remarquons que le `glue_channel` est assez insensible à la variation de la taille des messages, contrairement aux canaux TF. Ceci peut s'expliquer par le fait que le `glue_channel` ne simule pas de latence de transfert, ce qui permet à un message, peu importe sa taille, de se rendre dans la liste interne au canal en un seul delta-cycle. Dans les deux autres cas, l'appel de la fonction `wait()` pour simuler la latence de transfert provoque inévitablement des changements de contexte dans le simulateur de SystemC, résultant en une baisse de performance.

4.4 Variantes dans l'implémentation

4.4.1 Types de processus

L'ensemble des résultats a montré que les performances d'une simulation avec SystemC diminuent significativement lorsque le nombre de modules augmentent. Dans nos bancs d'essais, tous les modules de l'usager contenaient un processus `SC_THREAD` ou `SC_CTHREAD`, ce qui consomme beaucoup de ressources dans le simulateur [20]. Une idée de raffinement serait de remplacer les processus perpétuels par des proces-

sus momentanés (`SC_METHOD`) dans les périphériques, les adaptateurs et les canaux, là où cela est possible. Ce remplacement ne peut pas se faire dans les modules de l'usager ; n'oublions pas que les `SC_METHOD` ne peuvent pas contenir de fonction d'attente dynamique (`wait`). En utilisant ce type de processus, notre plate-forme serait plus adaptée à la synthèse logique étant donné que les `SC_METHOD` semblent être les seuls types de processus supportés par certains compilateurs SystemC [37].

Un changement dans la méthodologie pourrait simplifier le passage du niveau UTF vers le niveau TF. En effet, nous pourrions demander à l'utilisateur de continuer d'utiliser des processus `SC_THREAD` au niveau TF et d'adapter avec un effort raisonnable notre architecture pour palier à ce changement. Dans la littérature [37], on prévoit peut-être éliminer les `SC_CTHREAD` de SystemC. Nous avions opté originalement pour les processus synchrones parce qu'ils semblaient bien modéliser ce que nous souhaitions. En plus d'assurer une meilleure probabilité de compatibilité avec les futures versions de SystemC, ce changement réduirait au minimum les modifications architecturales nécessaires pour changer de niveau d'abstraction.

4.4.2 Optimisations dans la communication

Une autre optimisation possible serait de remplacer autant que possible la synchronisation sur l'horloge par de la synchronisation par événements. Par exemple, les adaptateurs, la minuterie et l'arbitre du bus contiennent des processus qui s'activent à chaque cycle d'horloge pour effectuer un traitement.

Nous avons déjà mentionné que la conception de SPACE était basée sur le fait que les modules de l'usager n'avaient qu'un seul processus communicant. Pour éliminer cette restriction, plusieurs modifications devraient être apportées, notamment au niveau des adaptateurs de module. Ces derniers connaissent le numéro

unique d'identification du module qui lui est connecté. De cette façon, le canal peut rediriger les messages vers le bon adaptateur assez facilement, en identifiant l'index du milti-port qui pointe vers l'adaptateur désiré, comme le montre ce bout de code simplifié du `space_channel_xbar` :

```
[...]
  unsigned long port_num = getAdapterNumber(TargetID);
  if (port_num != SOFTWARE)
  {
    // Adaptateur correspondant trouvé
    m_AdapterPorts[port_num]->write_from_channel(...);
  }
  else
  {
    // On peut conclure qu'il s'agit d'un module logiciel
    m_AdapterPorts[getAdapterNumber(0)]->write_from_channel(...);
  }
[...]
```

Cet algorithme devrait être modifié. Une solution serait de dresser automatiquement une liste des ID des modules dans l'adaptateur en début de simulation. Les adaptateurs pourraient ensuite envoyer ces listes au canal et un système de routage plus perfectionné qu'actuellement pourrait assurer la communication durant la simulation. Cette technique permettrait non seulement d'avoir des modules usagers multi processus (multithreaded), mais aussi de créer des hiérarchies de canaux à plusieurs niveaux. La communication logicielle/matérielle pourrait aussi profiter de cet ajout fonctionnel pour permettre d'étendre la plate-forme à plusieurs processeurs.

Les résultats de la figure 4.8 nous ont permis de montrer que le temps de recherche des requêtes dans les listes peut devenir un facteur important de réduction de performance. Tous les périphériques, adaptateurs et canaux utilisent des listes pour stocker les requêtes et les messages et à notre avis il serait possible d'utiliser d'autres structures qui accélèreraient la recherche et donc la simulation. Une étude de ce qu'offre STL par exemple pourrait nous permettre de construire quelques bancs d'essais et par analyse de performance, nous pourrions remplacer les listes par d'autres structures plus appropriées comme des arbres ou des tables de hachage. Nous avons opté pour la simplicité des listes dans notre première implémentation, car la performance n'était pas le but visé par notre premier prototype.

4.4.3 Fonctionnalités des périphériques

En plus d'optimiser les canaux de communication, nous pourrions améliorer les performances et les fonctionnalités de SPACE en optimisant les périphériques. Par exemple, le gestionnaire d'interruption a été conçu très simplement, mais il serait possible de réduire la latence des interruptions du RTOS en incorporant l'algorithme d'arbitrage des interruptions dans le gestionnaire matériel. Une façon rapide et efficace de procéder serait de traiter les interruptions en prenant comme priorité leur numéro d'IRQ. Cette solution simplifie les choses, mais ne permet pas d'avoir des priorités dynamiques ni de partager une même priorité à deux périphériques différents. Il serait aussi possible d'y ajouter plusieurs autres fonctionnalités, comme un masque des interruptions par exemple. Nous avons voulu garder le périphérique simple et ces modifications pourront être effectuées ultérieurement si cela est jugé nécessaire.

Concernant la minuterie, pour l'instant il n'est pas possible de questionner le périphérique de quelque façon que ce soit pour lui demander s'il est actif ou non,

pour la simple et bonne raison que cette fonctionnalité a été jugée impertinente pour le moment. Nous pourrions également imaginer d'autres modes de fonctionnement ; ils n'auront qu'à être ajoutés si l'utilisation de la minuterie démontre un manque de fonctionnalités. Finalement, l'ajout d'une seconde minuterie de type "one shot" pourrait être intéressant.

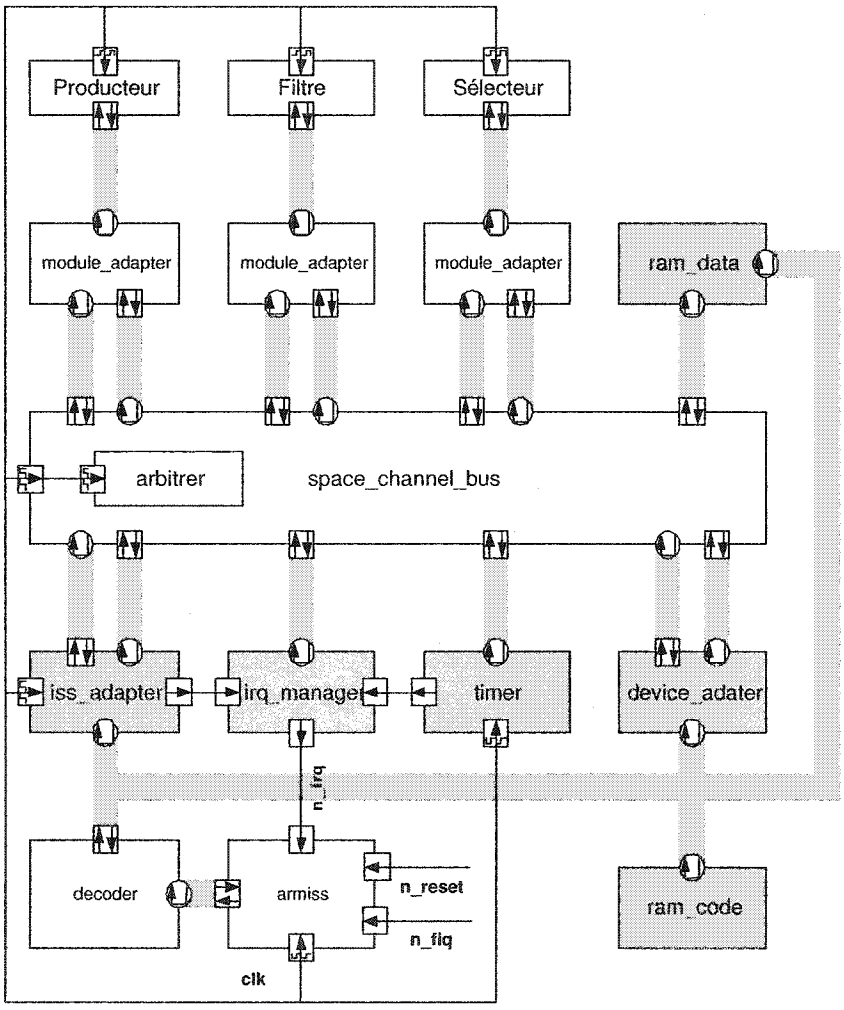

FIGURE 4.5 Exemple partitionné avec le bus

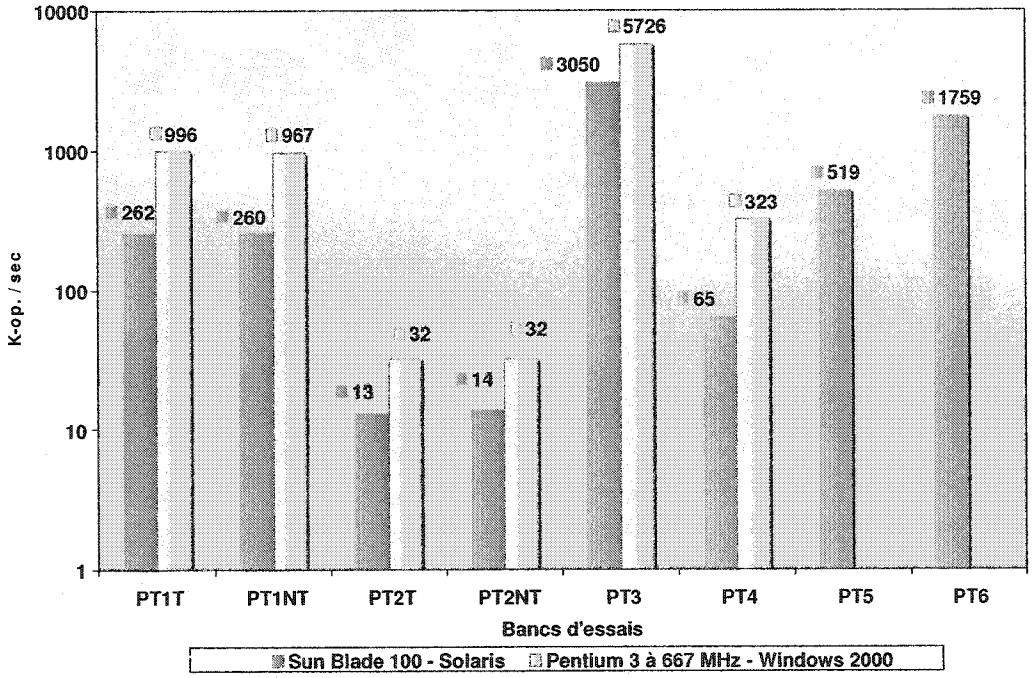

FIGURE 4.6 Performances comparatives SPACE / SOCP / Simple Bus

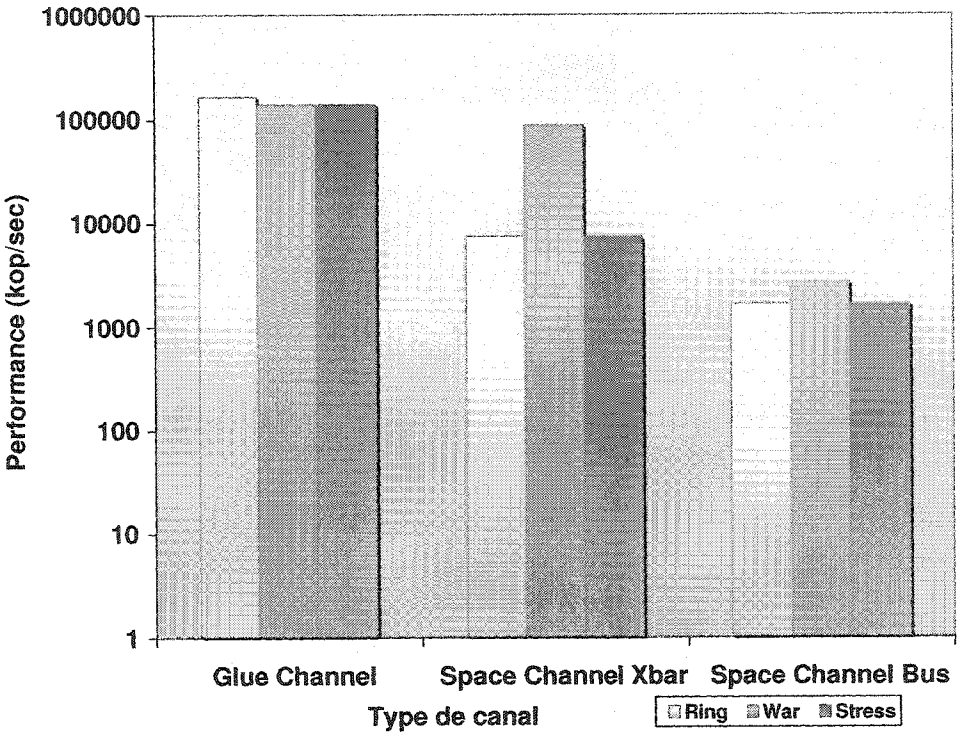

FIGURE 4.7 Performances paramétriques, cas #1

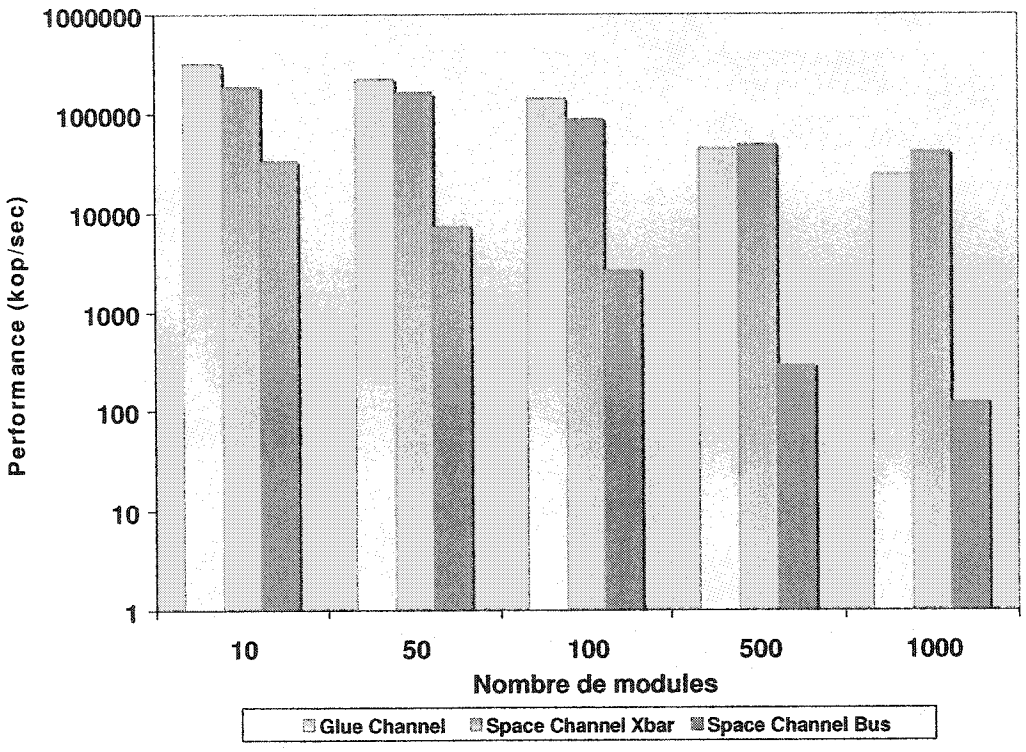

FIGURE 4.8 Performances paramétriques, cas #2

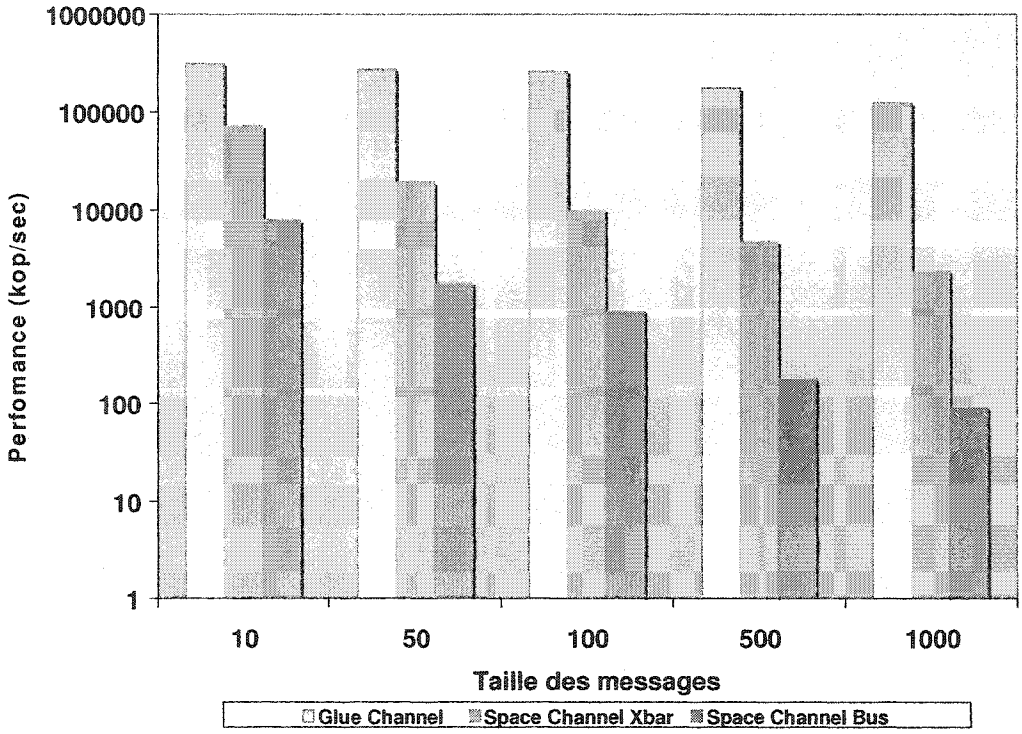

FIGURE 4.9 Performances paramétriques, cas #3

CONCLUSION

Au courant des dernières années, les avancements technologiques dans le domaine des semi-conducteurs, particulièrement au niveau de la densité des transistors, ont permis la conception de systèmes sur puce de plus en plus complexes. Ces derniers contiennent plusieurs blocs logiques, tels des processeurs, des bus et de la mémoire, que l'on retrouvait habituellement sur une carte et qui peuvent facilement contenir quelques dizaines de millions de transistors. De nos jours, les systèmes embarqués sont de plus en plus réalisés sur des systèmes sur puce (SOC). Cette nouvelle réalité, combinée aux pressions toujours présentes d'augmenter la productivité et de diminuer le temps de mise en marché, a favorisé l'apparition de nouvelles méthodes de conception, visant à rehausser le niveau d'abstraction des spécifications et favorisant le développement de prototypes fonctionnels lors de la phase de conception. SystemC est actuellement le langage le plus populaire concernant la spécification unifiée des systèmes numériques logiciels/matériels.

Notre projet nommé SPACE veut fournir une solution aux lacunes de SystemC 2.0 concernant la modélisation logicielle. Un concepteur peut ainsi utiliser notre plateforme et concevoir une application logicielle/matérielle en SystemC et simuler avec plus de précision les deux parties simultanément. Ce mémoire a présenté une partie du projet SPACE, c'est-à-dire l'implémentation des communications sur la plateforme. L'avantage majeur de SPACE est sans doute ses niveaux d'abstraction, qui permettent au concepteur de choisir le niveau de détails voulu pour ses simulations. La principale lacune est le fait qu'il n'y ait pas vraiment d'implémentation RTL équivalente au modèle raffiné. Citons également que nos interfaces ne correspondent pas vraiment à un standard existant, comme SOCP par exemple. Enfin, comme nous l'avons déjà mentionné, la synchronisation est trop souvent basée sur un signal d'horloge et non basée sur des événements dynamiques.

Nous avons implémenté la plate-forme avec comme premier objectif son fonctionnement, il est donc évident que plusieurs améliorations pourraient y être apportées afin de rendre de meilleures performances en simulation et pour faire de SPACE un outil de codesign plus complet. Comme nous en avons déjà discuté, le raffinement du modèle de communication pourrait être un bon point de départ. À notre avis, la première fonctionnalité à ajouter serait le support des multi processus au niveau des interfaces de communication. Concrètement, cela impliquerait que les méthodes des interfaces devraient contenir des identificateurs (comme des numéros par exemple) pour spécifier le processus d'origine et celui de destination. Pour l'instant il est seulement possible d'adresser un certain module usager, sans pouvoir adresser individuellement tel ou tel processus qu'il contient. En ajoutant cette caractéristique, il faudrait inévitablement changer l'architecture de SPACE, notamment les adaptateurs de modules et les canaux pour implémenter un algorithme de routage de messages plus perfectionné. Toutefois, après avoir effectué ces modifications, SPACE pourrait supporter de nombreuses autres fonctionnalités sans trop d'efforts de conception, comme la possibilité d'instancier plusieurs canaux et processeurs dans la même simulation. En effet, la plate-forme ne supporte pas plusieurs instances d'ISS dans la même simulation et ceci constitue une restriction qui va à l'encontre de la tendance actuelle des systèmes sur puces : ces derniers contenant de plus en plus d'unités de calcul RISC sur la même puce.

En ayant l'opportunité d'instancier plusieurs canaux, nous pourrions créer un périphérique spécial pour les relier, sous forme d'un adaptateur. De plus, le développement d'un autre type d'adaptateur, qui possède les interfaces SPACE d'un côté et les ports et interfaces de SOCP de l'autre, pourrait nous permettre de connecter les applications de l'usager sur la plate-forme StepNP.

Pour l'instant, le concepteur qui utilise SPACE peut se fier aux résultats de simulation pour obtenir le temps simulé (en cycles) afin de guider son choix de

partitionnement. Bien que le code des modules de l'usager n'ait pas à être modifié pour passer d'une configuration architecturale à une autre, l'utilisateur doit tout de même refaire de nouveaux fichiers qui définissent les connexions logicielles et matérielles de chacune de ces configurations. Cette opération prend un certain temps qui peut être non négligeable, car elle est propice à l'erreur d'inattention. Un programme informatique simple pourrait être conçu pour faciliter cette tâche qu'est le branchement des instances de modules et de périphériques pour instancier une nouvelle plate-forme SPACE : nous avons envisagé le développement d'un outil (éventuellement avec support graphique) pour assister le concepteur dans ses changements architecturaux. Un tel outil pourrait ainsi rehausser le niveau de détail des modifications à effectuer. L'utilisateur pourrait rapidement choisir son partitionnement, les adresses des périphériques et les connexions désirées et l'outil pourrait générer les fichiers d'élaboration et effectuer des vérifications simples sur la configuration, comme par exemple vérifier que les ID des modules sont uniques ou encore s'assurer que les périphériques respectent les règles d'adressage (celles que nous avons présentées à la figure 2.4).

En plus des cycles d'horloges, nous avons vu que plusieurs autres informations sont accessibles au concepteur en fin de simulation, comme la quantité de mémoire requise dans les adaptateurs de module, le temps d'attente des processus après l'arbitre ou encore le pourcentage d'utilisation du bus. D'autres critères d'évaluation, en plus de ceux qui concernent la performance, devraient être considérés. Il serait possible de faire une évaluation de la consommation de puissance à haut niveau. L'évaluation pourrait être statique, c'est-à-dire que les calculs seraient basés sur le choix des composants qui participent à la simulation, comme le type de canal choisi, le nombre de modules ou d'adaptateurs. À cette analyse pourrait s'ajouter un mécanisme d'évaluation de la puissance dynamique. Nous pourrions imaginer inclure des méthodes d'évaluation de puissance dans chaque module ou périphérique

[8]. Chaque entité de la plate-forme pourrait alors envoyer de l'information sur la puissance dissipée à un noyau central.

Bien que nous travaillons au niveau TLM et non au niveau RTL, le temps d'exécution pour les simulations peut tout de même s'avérer relativement long, plus particulièrement pour celles qui impliquent l'utilisation de l'ISS qui est précis au cycle près. N'oublions pas que ce dernier est précis au niveau des instructions et cela consomme beaucoup de ressources en simulation. Si dans les versions futures de SPACE nous désirons supporter plusieurs instances de processeurs dans une même architecture, il faudra songer à une solution pour distribuer la simulation sur plusieurs postes de travail. Un prototype nommé "dsim" a déjà été conçu pour SOCP [40] et une implémentation semblable dans SPACE pourrait être réalisée. Une simulation distribuée apporte malheureusement des temps de transfert non négligeables et il faut donc éviter de distribuer des tâches qui communiquent fréquemment.

Toutes ces modifications et ajouts concernent les niveaux existants UTF et TF pour lesquels la plate-forme a été conçue. Un raffinement intéressant dans notre méthodologie serait d'y ajouter d'autres niveaux d'abstraction. Le niveau sous-jacent au niveau TF serait le plus intéressant. Le raffinement matériel implique la transformation des mécanismes TLM vers le RTL. Les interfaces de communication doivent alors être remplacées par un ensemble de signaux et de ports élémentaires. Ceci impliquerait aussi de raffiner notre méthode d'adressage ID. Une façon de faire est d'assigner des adresses non utilisées aux modules et ainsi de faire disparaître les numéros d'identification au profit des adresses mémoire, plus appropriées à l'implémentation physique finale. De plus, tous nos mécanismes qui impliquent des listes STL (i.e. les listes de requêtes) devront être raffinés, car cela ne reflète pas une implémentation matérielle, mais constitue plutôt une astuce de modélisation abstraite. Ces listes pourraient devenir des mémoires de messages, formées d'un contrôleur et d'un espace de stockage.

RÉFÉRENCES

[1] ARM LIMITED. [En ligne]. http ://www.arm.com/. (Page consultée le 15 janvier 2004).

[2] ARM LIMITED. *AMBA Specification (Rev. 2.0).* 1999.

[3] BENINI, L., BERTOZZI, D., BRUNI, D., DRAGO, N., FUMMI, F. ET PONCINO, M. *Legacy SystemC Co-Simulation of Multi-Processor Systems-on-Chip.* IEEE International Conference on Computer Design : VLSI in Computers and Processors, 2002.

[4] BENNY, O., RONDONNEAU, M., CHEVALIER, J., BOIS, G., ABOULHAMID, E.-M. ET BOYER, F.-R. *SoC software refinement approach for a SystemC platform.* Design & Verification Conference & Exhibition (DVCon), 2004.

[5] BERTOLA, M. ET BOIS, G. *A methodology for the design of AHB bus master wrappers.* Euromicro Symposium on Digital Systems Design, Turquie, Septembre 2003.

[6] BESANA, M. ET BORGATTI, M. *Application Mapping to a Hardware Platform through Automated Code Generation Targeting a RTOS : A Design Case Study.* Design, Automation and Test in Europe Conference and Exhibition Design Forum, Allemagne, Mars 2003.

[7] BOIS, G., FILION, L., TSIKHANOVICH, A. ET ABOULHAMID, E. *Modélisation, raffinement et techniques de programmation orientée objet avec SystemC.* Publication en cours, 2004.

[8] CALDARI, M., CONTI, M., COPPOLA, M., CRIPPA, P., ORCIONI, S., PIERALISI, L. ET TURCHETTI, C. *System-Level Power Analysis Methodology Applied to the AMBA AHB Bus.* Design, Automation and Test in Europe Conference and Exhibition, Mars 2003.

[9] CHEVALIER, J. *Partitionnement, estimation et raffinement de système logiciel/matériel conçu à haut niveau en SystemC.* École Polytechnique de Montréal, Montréal, Thèse en préparation.

[10] CHEVALIER, J., RONDONNEAU, M., BENNY, O., BOIS, G., ABOULHAMID, E.-M. ET BOYER, F.-R. *SPACE : A Hardware/Software SystemC modeling platform including an RTOS.* Forum on specification & Design Languages (FDL), 2003.

[11] CYR, G. *Interface configurable pour un processeur ARM basée sur le protocole VCI.* Université de Montréal, Montréal, Février 2001.

[12] DE KOCK, E. A., ESSINK, G., SMITS, W. J. M., VAN DER WOLF, P., BRUNEL, J.-Y., KRUIJTZER, W. M., LIEVERSE, P. ET VISSERS, K. A. *YAPI : Application Modeling for Signal Processing Systems.* Design Automation Conference, 2000.

[13] DEITEL, H. M. ET DEITEL, P. J. *C++ How to Program, 4/E.* Prentice Hall, Décembre 2002.

[14] FILION, L., BOIS, G. ET ABOULHAMID, E. M. *Syslib : A System-Level Library Extended from Cynlib for SoC.* HDL Conference and Exhibit, États-Unis, 2002.

[15] FORTE DESIGN SYSTEMS. *Cynlib Language Reference Manual.* [En ligne]. CynApps Suite Release 1.0, http ://www.cynapps.com/. (Page consultée le 15 janvier 2004).

[16] FORTE DESIGN SYSTEMS. *Cynlib Users Manual.* [En ligne]. CynApps Suite Release 1.0, http ://www.cynapps.com/. (Page consultée le 15 janvier 2004).

[17] GAILHARD, S., INGREMEAU, O., DIGUET, J.-P., JULIEN, N. ET MARTIN, E. *Une méthode probabiliste pour estimer la consommation à un niveau algorithmique.* Colloque CAO circuits et systèmes, Villars de Lans, 1997.

[18] GERSTLAUER, A., YU, H. ET GAJSKI, D. *RTOS Modeling for System Level Design.* Design, Automation and Test in Europe Conference, Mars 2003.

[19] GNU. [En ligne]. http ://www.gnu.org/. (Page consultée le 15 janvier 2004).

[20] GRÖTKER, T., LIAO, S., MARTIN, G. ET SWAN, S. *System Design with SystemC.* Kluwer Academic Publishers, États-Unis, Mai 2002.

[21] HAVERINEN, A., LECLERCQ, M., WEYRICH, N. ET WINGARD, D. *White Paper for SystemC based SoC Communication Modeling for the OCP Protocol.* Octobre 2002.

[22] HOMMAIS, D. *Une méthode d'évaluation et de synthèse des communications dans les systèmes intégrés matériel-logiciel.* Université de Paris VI, France, Septembre 2001.

[23] IEEE. *IEEE Standard Verilog Hardware Description Language.* [En ligne]. http ://www.verilog.com/IEEEVerilog.html. (Page consultée le 15 janvier 2004).

[24] IEEE. *IEEE Standard VHDL Language Reference Manual.* IEEE Computer Society Press, 1988.

[25] KAHN, G. *The Semantics of a Simple Language for Parallel Programming.* IFIP Congress, 1974.

[26] LABROSSE, J. J. *MicroC/OS-II, the Real-Time Kernel, Second Edition.* CMP Books, 2002.

[27] LAURENT, J., JULIEN, N. ET MARTIN, E. *High Level Power Analysis for Embedded DSP Software.* IEEE TCCA Newsletter, 2001.

[28] MENTOR GRAPHICS. *ModelSim.* [En ligne]. http ://www.model.com/, 2003. (Page consultée le 15 janvier 2004).

[29] MENTOR GRAPHICS. *Seamless CVE.* [En ligne]. http ://www.mentor.com/seamless/, 2003. (Page consultée le 15 janvier 2004).

[30] MUELLER, W., RUF, J., HOFFMANN, D., GERLACH, J., KROPF, T. ET ROSENSTIEHL, W. *The Simulation Semantics of SystemC.* Design, Automation, and Test in Europe, Allemagne, Mars 2001.

[31] NICOLESCU, G., YOO, S., BOUCHHIMA, A. ET JERRAYA, A. A. *Validation in a component-based design flow for multicore SoCs.* ACM Press, 2002.

[32] NUTT, G. *Operating Systems, A Modern Perspective, Second Edition.* Addison Wesley, 2000.

[33] OCP-IP ASSOCIATION. *Open Core Protocol Specification.* 2001.

[34] ON-CHIP BUS DEVELOPMENT WORKING GROUP. *Virtual Component Interface Standard (OCB 2 2.0).* VSI Alliance, Août 2000.

[35] OSCI. *Functional Specification for SystemC 2.0.1.* [En ligne]. Open SystemC Initiative, http ://www.systemc.org/, 2002. (Page consultée le 15 janvier 2004).

[36] OSCI. *SystemC Version 2.0.1 User's Guide.* [En ligne]. Open SystemC Initiative, http ://www.systemc.org/, 2002. (Page consultée le 15 janvier 2004).

[37] OSCI. *SystemC 2.0.1 Language Reference Manual Revision 1.0.* [En ligne]. Open SystemC Initiative, http ://www.systemc.org/, 2003. (Page consultée le 15 janvier 2004).

[38] OUSSOROV, I., RAAB, W., HACHMANN, U. ET KRAVTSOV, A. *Integration of Instruction Set Simulators into SystemC High Level Models.* Euromicro Symposium on Digital System Design, 2002.

[39] PASRICHA, S. *Transaction level modeling of SoC with SystemC 2.0.* Synopsys User Group Conference, Inde, Mai 2002.

[40] PAULIN, P. G., PILKINGTON, C. ET BENSOUDANE, E. *StepNP : A System-Level Exploration Platform for Network Processors.* IEEE Design and Test of Computers, Novembre 2002.

[41] PAULIN, P. G., PILKINGTON, C., BENSOUDANE, E. ET LANGEVIN, M. *Domain-specific Multi-Processor SoC's : Platforms, Tools and Methods.* International Workshop on Software and Compilers for Embedded Systems, Australie, Septembre 2003.

[42] RONDONNEAU, M. *Intégration d'un RTOS dans une plate-forme SystemC destinée à l'exploration architecturale.* École Polytechnique de Montréal, Montréal, Mémoire en préparation.

[43] ROWSON, J. *Hardware-software co-simulation.* Design Automation Conference, Juin 1994.

[44] SCHWARTZ, K. *Modeling with SystemC 2.0.* International HDL Conference and Exhibition, Mars 2002.

[45] SWAN, S. *An Introduction to System Level Modeling in SystemC 2.0.* Open SystemC Initiative, http ://www.systemc.org/ [En ligne], Mai 2001. (Page consultée le 15 janvier 2004).

[46] TANENBAUM, A. S. *Modern Operating Systems.* Prentice Hall, 1992.

[47] WIND RIVER. *VxSim Datasheets.* [En ligne]. http ://www.windriver.com/products/vxsim/, 2003. (Page consultée le 15 janvier 2004).

[48] WIND RIVER. *VxWorks 5.x Datasheet.* [En ligne]. http ://www.windriver.com/products/vxworks5/, 2003. (Page consultée le 15 janvier 2004).

[49] ZIVOJNOVI, V. ET MEYR, H. *Compiled hardware-software co-simulation.* Design Automation Conference, Juin 1996.

ANNEXE I

CO-SIMULATION DU MATÉRIEL ET DU LOGICIEL

I.1 Motivations

SystemC inclut un simulateur qui s'apparente beaucoup à d'autres simulateurs matériels, comme les outils de simulations pour le VHDL par exemple [28]. Dans le même ordre d'idées, il existe également des simulateurs pour le logiciel. Ces simulateurs permettent d'exécuter du code logiciel qui suit une philosophie temps réel et qui utilise des constructions propres à un RTOS choisi (par exemple les outils de WindRiver [48, 47]). Pour exécuter la partie logicielle comme faisant partie d'un système complet, on peut décider d'inclure ou non un modèle de processeur. Dans le dernier cas, l'exécution est plus rapide puisqu'elle s'effectue sur la machine hôte directement. Nous voyons que les deux types de simulateurs, soit matériels et logiciels, ne sont pas conçus de la même façon et utilisent des techniques différentes pour simuler du code qui de toute manière modélisent des systèmes de natures différentes. Un problème se pose lorsque nous désirons effectuer une simulation d'un système composé de parties à la fois matérielles et logicielles [43, 49].

Il existe plusieurs façons de faire de la co-simulation. Une première façon est de prendre les deux types de simulateurs et d'essayer de les connecter ensemble, de les synchroniser, comme c'est le cas pour l'outil Seamless CVE de Mentor Graphics [29].

Une autre façon de faire, soit l'approche de co-simulation avec SystemC, est tout simplement de fusionner les deux simulateurs en un seul. La co-simulation est donc

une simple simulation. Ceci est possible, car le langage de description est le même. Par contre, pour bien respecter le comportement du logiciel, il faudrait ajouter des éléments de construction logiciels dans SystemC ou encore modifier l'ordonnanceur pour qu'il se comporte différemment avec les processus logiciels.

Notre approche vise un peu à tirer avantage de ces deux méthodes, afin d'une part de séparer les simulateurs pour mieux modéliser les comportements différents du logiciel et du matériel, tout en conservant le même langage de description pour les deux parties, soit du code C++ lié à SystemC.

Il y a plusieurs problèmes à résoudre pour pouvoir effectuer correctement une co-simulation logicielle/matérielle. Nous allons ici décrire brièvement les principaux problèmes, car notre solution doit les affronter.

I.2 Parallélisme versus concurrence

En matériel, les processus s'exécutent généralement en parallèle, tandis qu'en logiciel ils s'exécutent de façon séquentielle. Avec l'utilisation d'un système d'exploitation, plusieurs processus peuvent être en cours d'exécution simultanément. Néanmoins, un seul est actif à la fois. On parle alors de concurrence pour une ressource partagée, qui est le processeur. Pour modéliser le parallélisme, on exécute plusieurs processus avant d'augmenter le temps de simulation, tandis que pour modéliser la concurrence, on doit augmenter le temps de simulation pour chaque processus exécuté.

I.3 Temps d'exécution zéro

Souvent pour modéliser une latence d'exécution, on exécute une série d'opérations d'un seul coup, et ensuite on spécifie le délai pour ce groupe d'opérations. Il faut alors s'assurer qu'on ne compromet pas la cohérence dans le calcul du temps de simulation et être sûr que le processus ne peut pas être interrompu entre le moment où l'exécution des opérations débute et le moment où le délai est spécifié. Aussi, à un certain niveau d'abstraction, on peut vouloir enlever le plus possible les notions de délai, souvent pour accélérer la simulation. Il faut alors recourir à l'utilisation des événements pour synchroniser le tout. Souvent en matériel on utilise le delta-cycle pour respecter quand même l'ordre d'exécution (voir plus loin pour un explication sur les delta-cycles).

I.4 Enchaînement d'événements

Un processus peut vouloir une donnée qui n'est pas disponible, ou encore doit attendre un certain moment venu pour s'exécuter. On utilise les événements pour synchroniser les processus et pour leur signaler des changements d'états externes. Si on connaît d'avance le temps d'attente nécessaire, on peut alors spécifier un délai et de façon implicite on met le processus en attente sur un événement de l'horloge. Les notions d'attente active et d'attente inactive entrent alors en compte. On dit qu'une attente inactive consiste à endormir un processus (le mettre hors exécution) jusqu'à ce qu'un événement se produise. Le processus est alors réveillé et l'attente, terminée. L'attente inactive est celle qui est toujours préconisée dans les simulateurs matériels. L'attente active, quant à elle, consiste à laisser le processus en exécution jusqu'à ce que le délai d'attente soit atteint. Cela ne fait pas vraiment de sens en matériel, mais est possible en logiciel. Au niveau de la simulation, l'attente active est

souvent transformée en attente inactive, c'est-à-dire que l'exécution du processus sera interrompue par l'arrivée d'un événement externe. On appelle souvent cette technique le pilotage événementiel [22].

Pour résumer, nous disons que les processus matériels doivent "rendre la main" pour informer le simulateur que leur exécution ponctuelle est terminée, tandis qu'en logiciel les processus peuvent soit rendre la main (système non préemptif) ou alors être interrompus par l'arrivée d'un événement externe (système préemptif). Plus de détails seront donnés plus loin, voir Ordonnancement.

I.5 Delta-cycles

Le concept de delta-cycle est devenu une connaissance triviale pour les concepteurs matériels. Le delta-cycle est une astuce de simulation pour permettre d'ordonnancer les processus par rapport à leurs dépendances de données ou d'événements. Pour un même cycle d'horloge fixe, le temps de simulation est figé, mais il peut y avoir plusieurs delta-cycles. Pour chacun de ces delta-cycles, plusieurs processus peuvent être exécutés. Ainsi, il est possible de respecter l'ordre d'exécution des processus (ordonnancement séquentiel) qui normalement sont simulés de façon parallèle.

Cela pose problème lors de l'affectation des variables partagées entre plusieurs processus. Par exemple, si plusieurs processus veulent modifier une certaine variable dans le même delta-cycle en lui affectant différentes valeurs, cette variable ne peut pas prendre toutes les valeurs de façon effective (séquentiellement), car les processus sont simulés de façon simultanée (parallèle). Cependant, pour résoudre l'affectation finale de la variable à la fin du cycle d'horloge, il faut garder un historique des affectations multiples attribuées à la variable lors de tous les delta-cycles du cycle courant.

La façon de résoudre le problème est de conserver toutes ces valeurs dans le simulateur de façon à avoir les valeurs actuelles et futures pour les variables partagées. Il est ainsi facile de voir que la notion de delta-cycle n'est pas censée pour la modélisation de logiciel. En logiciel, les variables partagées doivent être affectées effectivement par tous les processus.

I.6 Variables et signaux

La notion de signal est propre au monde matériel. L'utilisation des delta-cycles, comme nous venons de le voir, est utilisée de paire avec les signaux dans les simulateurs matériels.

Les signaux non plus n'ont pas vraiment d'équivalent en logiciel [45]. D'abord, les signaux ont des valeurs logiques qui représentent des états électriques (comme "X" ou "Z") en plus des valeurs logiques ("0" ou "1"). Ensuite, les signaux peuvent être pilotés par plusieurs sources et une fonction de résolution peut alors calculer la valeur effective du signal. En logiciel, une variable qui est modifiée par plusieurs processus changera tout simplement de valeur. Finalement, comme les signaux sont compatibles aux environnements de simulation avec des delta-cycles, l'affectation de nouvelles valeurs ne prend pas effet immédiatement. Cela permet entre autre de modéliser deux registres qui s'échangent leurs valeurs sur un coup d'horloge. En logiciel, l'intervention d'une troisième variable temporaire est indispensable pour effectuer une telle opération.

I.7 Cohérence

Lorsqu'une co-simulation est effectuée au moyen de deux simulateurs distincts, certaines informations dans un simulateur peuvent vouloir être connues dans l'autre. L'état interne d'un simulateur est donc l'état externe de l'autre. Certaines parties de l'état interne des simulateurs doit pouvoir être accessible de l'extérieur pour assurer la cohérence. Un exemple typique est celle d'une mémoire qui est partagée entre le logiciel et le matériel.

I.8 Synchronisation logicielle/matérielle

La synchronisation entre les deux simulateurs est aussi un problème qu'il faut résoudre. Une approche simple est de faire avancer les deux simulateurs cycle par cycle. Évidemment cela engendre un coût assez important en temps de synchronisation. Une façon plus performante de procéder est de fonctionner sur événements. Les simulateurs se synchronisent alors que sur des changements d'états qui impliquent le logiciel et le matériel. Dans le cas où le simulateur logiciel (ISS, [38]) est précis au cycle près, la synchronisation par événements revient à la synchronisation au cycle près. La synchronisation logicielle/matérielle se complique davantage lorsque les deux simulateurs doivent fonctionner dans des domaines d'horloge différents, i.e. à des fréquences d'opérations différentes.

I.9 Communication logicielle/matérielle

Pour assurer la communication logicielle/ matérielle, il y a plusieurs façon de procéder. Il est possible de recourir à des techniques assez simples qui consistent à partager des espaces mémoires d'un simulateur à un autre. Certaines astuces n'ont

pas vraiment de sens au niveau modélisation ; elles ne font que permettre une simulation fonctionnelle. Il peut être intéressant d'avoir recours à des mécanismes qui pourront éventuellement être réutilisés durant le raffinement, donc qui s'apparentent plus à la fonctionnalité que l'on veut modéliser. Dans ce cas, la communication logicielle/matérielle fait donc partie du modèle et non pas seulement des simulateurs.

En logiciel, la communication fonctionne habituellement avec des queues de messages, ou encore avec des variables partagées qui sont protégées par des sections critiques. En matériel, la communication peut se faire à petite échelle à l'aide de registres, mémoires tampons duales, queues de messages matérielles et à plus grande échelle à l'aide de protocoles de bus ou de réseaux sur puce. Les mécanismes les plus souvent utilisés pour l'intercommunication logicielle/matérielle sont les queues de messages parce qu'elles ont un équivalent dans les deux mondes. De plus, on tente souvent de remplacer les mécanismes d'attente active (polling) par une communication par interruptions (pour le logiciel) ou par signaux (pour le matériel).

I.10 Ordonnancement

Au niveau matériel, lorsque plusieurs processus ont à être exécutés dans le même delta-cycle, les simulateurs ont l'habitude d'exécuter les processus selon un certain ordre souvent issu de la structure interne du simulateur, par exemple provenant de la phase d'initialisation du simulateur, lors de la création des listes de sensibilités. Le résultat est que l'ordre d'exécution des processus est déterminé par le simulateur et non par la spécification. Ceci est parfois corrigé par un ordonnancement basé sur l'exécution aléatoire. Le rôle du simulateur est toutefois de rendre transparent l'ordre d'exécution, car la simulation doit respecter la cohérence des entrées/sorties des processus, alors que leur exécution interne reste privée. Après

tout, la spécification de processus parallèles évite de dicter un ordre d'exécution quelconque.

Au niveau du logiciel, sur un même processeur, nous savons que l'exécution des processus doit se faire de façon séquentielle. À l'aide d'un système d'exploitation temps réel, nous pouvons par exemple choisir entre l'ordonnancement de type tourniquet, FIFO de priorités, préemptif basé sur les priorités, etc. [32]

Au niveau de la co-simulation, l'ordonnancement du logiciel et du matériel doit évidemment être respecté, c'est pourquoi nous disons que les algorithmes d'ordonnancement du matériel ne sont pas appropriés pour le logiciel.

ANNEXE II

DÉTAILS DES RÉSULTATS EXPÉRIMENTAUX

II.1 Tests de performance comparatifs

Les tableaux II.1 et II.2 présentent les résultats des tests comparatifs entre plusieurs implémentations de TLM : SPACE, SOCP et Simple Bus. Dans ces tableaux, k-op/sec est un raccourci pour milliers d'opérations d'entrée/sortie par seconde et N.D. signifie que les résultats ne sont pas disponibles.

II.1.1 Résultats sur un Pentium III 667 MHz

TABLEAU II.1 Performances comparatives (PIII-667 MHz)

Cas	**Description**	**Transactions** (nombre)	**Exécution** (sec)	**Performance** (k-op/sec)
PT1T	X-Bar (avec processus)	10105740	10,14	996
PT1NT	X-Bar (sans processus)	9571480	9,89	967
PT2T	Bus (avec processus)	326898	9,96	32
PT2NT	Bus (sans processus)	325935	10,10	32
PT3	Glue Channel	56716253	9,90	5726
PT4	Simple Bus	3254140	10,06	323
PT5	SOCP fonctionnel	N.D.	N.D.	N.D.
PT6	SOCP sans canal	N.D.	N.D.	N.D.

II.1.2 Résultats sur un Sun Blade 100

TABLEAU II.2 Performances comparatives (Sun Blade 100)

Cas	**Description**	**Transactions** (nombre)	**Exécution** (sec)	**Performance** (k-op/sec)
PT1T	X-Bar (avec processus)	2571537	9,79	262
PT1NT	X-Bar (sans processus)	2604960	10,01	260
PT2T	Bus (avec processus)	137289	9,89	13
PT2NT	Bus (sans processus)	144540	10,31	14
PT3	Glue Channel	30959784	10,15	3050
PT4	Simple Bus	659559	10,00	65
PT5	SOCP fonctionnel	N.D.	9,93	519
PT6	SOCP sans canal	N.D.	10,07	1759

II.2 Tests de performance paramétriques

II.2.1 Cas #1

Les paramètres sont :

- Type d'application versus performance ;
- Type de canal versus performance ;
- Invariants : Modules = 100, Taille = 1.

TABLEAU II.3 Performances paramétriques, cas #1

Application	**Type de Canal**	**Performance** (k-op/sec)
Ring	Glue Channel	164861
Ring	Space Channel X-bar	7432
Ring	Space Channel Bus	1697
War	Glue Channel	141045
War	Space Channel X-bar	88515
War	Space Channel Bus	2731
Stress	Glue Channel	139735
Stress	Space Channel X-bar	7432
Stress	Space Channel Bus	1687

II.2.2 Cas #2

Les paramètres sont :

- Nombre de modules versus performance ;
- Type de canal versus performance ;
- Invariants : Taille = 1, Application = WAR.

TABLEAU II.4 Performances paramétriques, cas #2

Type de Canal	**Nombre de modules**	**Performance** (k-op/sec)
Glue Channel	10	322983
Glue Channel	50	224888
Glue Channel	100	141045
Glue Channel	500	44601
Glue Channel	1000	24010
Space Channel X-bar	10	189665
Space Channel X-bar	50	163450
Space Channel X-bar	100	88515
Space Channel X-bar	500	49716
Space Channel X-bar	1000	40905
Space Channel Bus	10	33634
Space Channel Bus	50	7457
Space Channel Bus	100	2731
Space Channel Bus	500	303
Space Channel Bus	1000	124

II.2.3 Cas #3

Les paramètres sont :

- Taille des messages versus performance ;
- Type de canal versus performance ;
- Invariants : Modules = 10, Application = WAR.

TABLEAU II.5 Performances paramétriques, cas #3

Type de Canal	**Taille des messages** (entiers de 32 bits)	**Performance** (k-op/sec)
Glue Channel	10	310579
Glue Channel	50	276244
Glue Channel	100	259717
Glue Channel	500	175147
Glue Channel	1000	122746
Space Channel X-bar	10	71933
Space Channel X-bar	50	19132
Space Channel X-bar	100	9953
Space Channel X-bar	500	4655
Space Channel X-bar	1000	2347
Space Channel Bus	10	7754
Space Channel Bus	50	1754
Space Channel Bus	100	893
Space Channel Bus	500	180
Space Channel Bus	1000	90

ANNEXE III

RÈGLES D'IMPLÉMENTATION DE SPACE

III.1 Composants réutilisables à instancier

Les utilisateurs de SPACE doivent se conformer aux règles de structurelles pour concevoir leurs propres blocs (modules ou périphériques) et aux règles architecturales afin de respecter la méthodologie et les limtes imposées de la plate-forme. Nous présentons ici ce qui peut constituer chacune des instances de SPACE.

III.1.1 Périphériques

L'assemblage de périphériques pré-conçus permet de construire rapidement une architecture. Voici une liste des périphériques avec le nombre d'instances minimales et maximales possibles pour chacun des périphériques entre crochets. Le paramètre n est un nombre entier strictement positif.

- Mémoire de code [0, n] (lorsque l'ISS est utilisé) ;
- Mémoire de données [0, n] ;
- Mémoire vidéo [0, n] ;
- Adaptateur du processeur [0, 1] (lorsque l'ISS est utilisé) ;
- Adaptateur des périphériques [0, 1] (lorsque l'ISS est utilisé) ;
- Minuterie [0, 1] (lorsque l'ISS est utilisé) ;
- Gestionnaire d'interruptions [0, 1] (lorsque l'ISS est utilisé) ;
- Périphérique d'arrêt [0, n] ;
- ISS [0, 1] ;

- Décodeur [0, 1] (lorsque l'ISS est utilisé) ;
- Adaptateur de module [1, n] (autant qu'il y a de modules, 1 minimum) ;
- Modules de l'usager [1, n] ;
- Canal [1] ;
- Plusieurs `SC_MODULE` bidons pour éviter les ports non connectés.

III.1.2 Signaux

Les signaux nécessaires sont toujours de type `sc_signal<bool>`, sauf pour le signal d'horloge, qui est de type `sc_clk`. Les nombres entre crochets représentent le nombre d'instances minimales et maximales possibles pour chacun des singaux.

- Horloge système [0, 1] (pour une simulation TF seulement) ;
- Signaux d'interruptions [0, 32 + 1] (là où nécessaire, TF seulement) ;
- Remise à zéro pour le processeur [0, 1] (lorsque l'ISS est utilisé) ;
- Signaux bidons pour éviter les ports non connectés.

III.1.3 Autres considérations

Une considération importance lors de l'instanciation des périphériques est le respect des règles d'adressage. Chaque périphérique peut occuper une quantité de mémoire précise dans la plage mémoire. Voici la quantité de mémoire occupée par chacun des périphériques (indiquée par l'expression entre crochets, où n est un entier positif).

- Mémoire de code [64kn] (plages de 64 Ko) ;
- Mémoire de données [64kn] (plages de 64 Ko) ;
- Mémoire vidéo [4] ;
- Adaptateur du processeur [4] ;

- Adaptateur des périphériques [4n] ;
- Minuterie [12] ;
- Gestionnaire d'interruptions [8] ;
- Périphérique d'arrêt [4].

Finalement, il y a d'autres paramètres à fixer pour instancier une plate-forme, soit :

- Le fichier binaire source pour la mémoire de code (si ce périphérique est utilisé) doit être construit pour former la partie logicielle ;
- La période de la minuterie (si ce périphérique est utilisé) doit être fixé ;
- Les ID des modules de l'application doivent être choisis.

III.2 Règles structurelles pour les modules

En plus de respecter les règles de configuration, le concepteur a la responsabilité de respecter les règles de codage et de conception, ainsi que les limitations de SPACE par rapport aux constructions de SystemC.

III.2.1 Niveau UTF

- L'application doit être partitionnée en un ou plusieurs éléments `SC_MODULE` ;
- Chacun des modules doit hériter d'une classe de base : `space_base_module`, définie pour le niveau UTF ;
- Chaque module peut contenir un ou plusieurs `SC_THREAD`. Si le module contient plusieurs `SC_THREAD`, chacun de ces processus doit communiquer avec d'autres modules différents (ou en d'autres mot, deux processus différents d'un même module ne peuvent pas communiquer avec un autre même module). Si le module possède au moins un processus, on doit inclure la macro `SC_HAS_PROCESS` dans

la définition de la classe du module ;

- Les modules doivent utiliser le port de communication unique fourni avec la classe de base pour communiquer avec d'autres modules ;
- Le concepteur peut utiliser des mots clés `wait` de SystemC. Cependant, s'il le fait, il doit utiliser la signature suivante :

  wait({nombre}, SC_NS) ;

  Cette forme de `wait` doit être remplacée ensuite par la signature suivante au niveau TF :

  wait({nombre}) ;

  Par conséquent, le concepteur doit utiliser la méthode `isHighLevel()` pour que son code ne soit pas modifié lors du passage UTF vers TF et vice-versa. Voici de quel façon il doit procéder :

```
if ( isHighLevel() )
    wait( {nombre}, SC_NS ); // valide pour les SC_THREAD
else
    wait( {nombre} ); // valide pour les SC_CTHREAD
```

III.2.2 Niveau TF

Les mêmes règles de codage s'appliquent sauf pour les exceptions qui suivent :

- Au lieu d'utiliser des `SC_THREAD`, le concepteur doit utiliser des `SC_CTHREAD` ;
- Au lieu d'hériter de la classe de base `space_base_module` définie pour le niveau UTF, les modules de l'usager doivent hériter de la classe de base `space_base_module` définie pour le niveau TF.

III.2.3 Structure des périphériques, niveaux UTF et TF

- L'usager peut concevoir lui-même des périphériques, s'il le fait il doit hériter d'une classe de base : `space_base_device` ;
- Le concepteur doit implémenter les méthodes :
  - `mem_write_from_bus()`
  - `mem_read_from_bus()`

  Ces méthodes seront appelées lorsqu'un accès en lecture ou en écriture sera initié vers le périphérique ;
- S'il le désire, le concepteur peut ajouter un port d'horloge à son périphérique. Dans ce cas il doit connecter ses instances de périphérique au signal d'horloge et il lui sera impossible de simuler son application au niveau UTF ;
- S'il le désire, le concepteur peut créer des processus dans son périphérique. Dans ce cas il doit inclure la macro `SC_HAS_PROCESS` dans la définition de la classe du périphérique. La méthode `isHighLevel()` lui permettra de faire des `wait` dans son processus ;
- Pour l'instant la structure logicielle de la plate-forme ne supporte pas l'ajout de modules ou de périphériques conçus par l'utilisateur qui utilisent les interruptions. Cela pourra éventuellement être ajouté comme une fonctionnalité disponible.

III.3 Règles architecturales

III.3.1 UTF et TF

- Le nombre d'instances différentes d'un même module n'est pas limité ;
- Chaque instance de module doit avoir un ID différent ;
- Le ID zéro est réservé pour l'adaptateur du processeur.

III.3.2 Règles architecturales UTF

- Le concepteur doit instancier un seul canal UTF ;
- Tous les ports de communication des modules doivent être connectés à l'interface du canal UTF ;
- Le multi-port des périphériques du canal UTF doit être connecté à l'interface de chacun des périphériques ;
- Dans le cas où le concepteur n'a pas besoin de périphérique, une instance de périphérique nul doit être créée et connectée au multi-port du canal UTF. Ce périphérique nommé `null_device` est simplement un `SC_MODULE` bidon.

III.3.3 Règles architecturales TF

- Le concepteur doit instancier un seul canal TF ;
- Il doit y avoir un signal d'horloge de période 1 ns, avec 50% de rapport de phase ;
- Pour chaque instance de module, le concepteur doit instancier un adaptateur de module ;
- Tous les modules de l'usager doivent être connectés à l'horloge et à leur adaptateur respectif ;
- Tous les adaptateurs de modules doivent être connectés à l'horloge et au canal TF ;
- Le canal TF doit être connecté au signal d'horloge si le type de canal possède un port d'horloge. Dans tous les cas, le multi-port des périphériques du canal TF doit être connecté à l'interface de chacun des périphériques et le multi-port des adaptateurs de module doit être connecté à l'interface de chacun des adaptateurs ;
- Tous les modules logiciels doivent être connectés à l'émulateur de SystemC (les ports de communication doivent être connectés à l'API) ;
- L'adaptateur du processeur et la minuterie sont des sources d'interruption. Il

faut brancher les signaux d'interruptions dans le gestionnaire d'interruption.

ANNEXE IV

EXEMPLES AVEC SYSTEMC

IV.1 Exemple 1

IV.1.1 Fichier prod.h

```
#include <systemc.h>

#ifndef PROD_H
#define PROD_H

SC_MODULE(prod)
{
    sc_fifo_out<int> port;

    void thread(void) {
        for (int i = 1; i<=10; i++) {
            port->write( i*100 );
        }
    }
    SC_CTOR(prod) {
        SC_THREAD(thread);
    }
};
```

```
#endif // PROD_H
```

IV.1.2 Fichier cons.h

```
#include <systemc.h>

#ifndef CONS_H
#define CONS_H

SC_MODULE(cons)
{
    sc_fifo_in<int> port;

    void thread(void) {
        int input;
        for (int i = 0; i<10; i++) {
            port->read( input );
            cout << "input = " << input << endl;
        }
    }
    SC_CTOR(cons) {
        SC_THREAD(thread);
    }
};

#endif // CONS_H
```

IV.1.3 Fichier top.h

```
#include "prod.h"
#include "cons.h"

#ifndef TOP_H
#define TOP_H

SC_MODULE(top)
{
    prod* prod1;
    cons* cons1;
    sc_fifo<int>* fifo1;

    SC_CTOR(top) {
        prod1 = new prod("prod1");
        cons1 = new cons("cons1");
        fifo1 = new sc_fifo<int>(1);

        prod1->port(*fifo1);
        cons1->port(*fifo1);
    }
};

#endif // TOP_H
```

IV.1.4 Fichier main.cpp

```
#include "top.h"

int sc_main(int argc, char**argv)
{
    top top1("top1");
    sc_start(-1);
    return 0;
}
```

IV.2 Exemple 2

IV.2.1 Fichier mychannel_if.h

```
#include <systemc.h>

#ifndef MYCHANNEL_IF_H
#define MYCHANNEL_IF_H

class mychannel_if: public sc_interface
{
public:
    virtual void mychannel_read(int& value) = 0;
    virtual void mychannel_write(int value) = 0;
};
```

```
#endif // MYCHANNEL_IF_H
```

IV.2.2 Fichier mychannel.h

```
#include "mychannel_if.h"

#ifndef MYCHANNEL_H
#define MYCHANNEL_H

class mychannel: public sc_channel, public mychannel_if
{
    int internal;
    bool empty;
    sc_event write_event;
    sc_event read_event;

public:
    mychannel(sc_module_name _name)
        : mychannel_if(), sc_channel(_name) {
        internal = 0;
        empty = true;
    }
    virtual void mychannel_read(int& value) {
        if (empty)
            wait(write_event);
        value = internal;
        empty = true;
        read_event.notify();
```

```
    }
    virtual void mychannel_write(int value) {
        if (!empty)
            wait(read_event);
        internal = value;
        empty = false;
        write_event.notify();
    }
};

#endif // MYCHANNEL_H
```

IV.2.3 Fichier prod.h

```
#include "mychannel_if.h"

#ifndef PROD_H
#define PROD_H

class prod : public sc_module
{
public:
    sc_port<mychannel_if> port;
    sc_in_clk clock;

    SC_HAS_PROCESS(prod);

    void cthread(void) {
```

```
        for (int i = 1; i<=10; i++) {
            port->mychannel_write( i*100 );
        }
    }
    prod(sc_module_name name_) : sc_module(name_) {
        SC_CTHREAD(cthread, clock.pos());
    }
};

#endif // PROD_H
```

IV.2.4 FIchier cons.h

```
#include "mychannel_if.h"

#ifndef CONS_H
#define CONS_H

class cons : public sc_module
{
public:
    sc_port<mychannel_if> port;
    sc_in_clk clock;

    SC_HAS_PROCESS(cons);

    void cthread(void) {
        int input;
```

```
        for (int i = 0; i<10; i++) {
            port->mychannel_read( input );
            cout << "input = " << input << endl;
        }
    }
    cons(sc_module_name name_) : sc_module(name_) {
        SC_CTHREAD(cthread, clock.pos());
    }
};

#endif // CONS_H
```

IV.2.5 FIchier top.h

```
#include "prod.h"
#include "cons.h"
#include "mychannel.h"

#ifndef TOP_H
#define TOP_H

class top : public sc_module
{
public:
    prod* prod2;
    cons* cons2;
    mychannel* channel2;
    sc_clock clk;
```

```
    top(sc_module_name name_) : sc_module(name_) {
        prod2 = new prod("prod2");
        cons2 = new cons("cons2");
        channel2 = new mychannel("channel2");

        prod2->port(*channel2);
        prod2->clock(clk);
        cons2->port(*channel2);
        cons2->clock(clk);
    }
};

#endif // TOP_H
```

IV.2.6 FIchier main.cpp

```
#include "top.h"

int sc_main(int argc, char**argv)
{
    top top2("top2");
    sc_start(-1);
    return 0;
}
```